ALGORITMOS E ESTRUTURAS DE DADOS

SORTING METHODS

ALEXANDRE SERRAS (97505) (1/3)

JOÃO REIS (98474) (1/3)

RICARDO RODRIGUEZ (98388) (1/3)

Turma P1

Prof. Tomás Oliveira e Silva

2020/2021

Índice

1. Introdução

Para a realização deste projeto, foi-nos fornecida uma pasta com vários ficheiros escritos em linguagem C, bem como alguns "*header files*" igualmente importantes.

O *sorting_methods.c* será o código principal. Este ficheiro vai utilizar uma série de funções para ordenar uma lista de dados, imprimindo o valor de n, os valores mínimos, médios e máximos de execução, bem como o desvio padrão de forma a poder analisar qual dos algoritmos de ordenação é o mais eficaz.

Ao longo deste relatório, vamos explicar o código do *sorting_methods.c*, explicar como funciona cada um dos algoritmos de ordenação utilizados e apresentar o respetivo gráfico de tempo de execução que o carateriza e, por último, vamos dizer qual das *sorting functions* é a mais indicada através da análise dos dados produzidos, onde vamos analisar as situações com arrays de tamanho 10, 100, 1000, 10000, 100000 e para cada uma delas vamos analisar os valores que foram obtidos e para concluir vamos analisar de uma forma mais geral os resultados obtidos.

2. Código sorting_methods.c

Tal como referido anteriormente, este ficheiro escrito em linguagem C é o "motor" do projeto, uma vez que é aquele que vai invocar cada algoritmo de ordenação e vai apresentar os respetivos valores de execução, fundamentais para posterior análise.

Inicialmente, chamamos uma série de bibliotecas de funções através do *"#include*": *math.h*, *time.h*, *stdio.h*, *stdlib.h*, *string.h*, bem como outros *header files* que não pertencem às bibliotecas padrão da linguagem C mas que estão presentes nas pastas fornecidas como, por exemplo, *sorting_methods.h* e *elapsed_time.h*.

A função *show* tem como argumentos de entrada um ponteiro para uma estrutura de dados e dois inteiros, *first* e *one_after_last*, não retornando qualquer tipo de valor. Esta função mostra, inicialmente, o par de inteiros introduzidos, seguindo-se um for loop que vai imprimir todos os valores da estrutura "*data*" como inteiros, terminando com o caractere especial "*n*" que marca o início de uma nova linha.

De seguida, encontra-se a função *main* com os parâmetros de entrada "*int argc*", um inteiro cujo valor é o número de argumentos introduzidos na linha de comando, e "*char *argv[argc]*", um ponteiro para um *array* de ponteiros que armazena os comandos introduzidos na execução do programa.

Dentro do corpo da função principal, temos uma declaração de um tipo de dados denominado de "*functions[]*". Esta *struct* tem dois atributos, uma variável "*function*" da classe *sort_function_t* e um ponteiro para uma *string* "*name*". Esta última variável pode ser expandida para assumir o nome do respetivo algoritmo de ordenação a ser usado.

A variável constante "*N_FUNCTIONS*" vai guardar um inteiro que corresponde ao número de bytes total do *array* "*functions*" sobre o valor de bytes que a primeira função ocupa (igual a todas as outras), ou seja, vai armazenar o número de funções disponíveis.

Se executarmos o programa com o segundo parâmetro de entrada igual a *"-test*", este vai verificar se todos os algoritmos estão a correr sem qualquer tipo de erro.

Define-se *MAX_N* com 1000 (testar *arrays* com este limite de tamanho) e *N_TESTS* com 100 (número limite de testes para cada *array* diferente). Inicializam-se as variáveis *i, j, k, n, first* e *one_after_last* como inteiros com o valor padrão, bem como dois *arrays* "*master*" e "*data*", ambos com tamanho *MAX_N*.

A linha de código "*srand((unsigned int)time(NULL))*" origina uma *seed* sempre diferente para o gerador de números pseudo-aleatórios, evitando a formação de números iguais.

Posteriormente, usa-se um for loop de n = 1 até n = MAX_N que vai formar um *array* com n elementos. Dentro deste, existe outro for loop cuja função é preencher a lista "*master*"

com valores aleatórios de 0 até 1000 (MAX_N) e, após este, guarda-se "*first = 0*" (primeiro *index* da lista) e "*one_after_last = n*" (número de elementos da lista).

Segue-se um for loop que vai realizar 100 testes (N_TESTS) para cada *array* e, dentro deste, testa-se, para cada uma das *sorting functions* existentes, se houve algum erro de acesso, imprimindo uma mensagem de erro e retornando o valor '1' caso isto se verifique.

Antes da verificação desta condição, todos os elementos de "*data*" das posições *first* até *one_after_last - 1* vão ter os mesmos valores que os de "*master*", sendo que os restantes guardam o valor '-1', e cria-se um novo objeto da *struct functions*[] com o nome *function* e com os atributos *data*, *first* e *one_after_last*.

Caso não se tenha verificado um erro de acesso, então verifica-se, para cada par de elementos adjacentes no *array data*, se o elemento anterior é maior que o elemento seguinte. Caso isto se verifique, executa-se a função *show* (já explicada anteriormente) e imprime-se um erro de ordenação, terminando o programa com *return* de '1'.

Antes de prosseguir para a próxima iteração, gera-se um número aleatório para a variável *first* e para *one_after_last*, sendo que o primeiro tem de ser menor ou igual que o segundo.

Termina assim o programa para quando introduzimos "*-test*", imprimindo no final a mensagem de sucesso e encerrando o programa sem qualquer tipo de erro.

No entanto, se a invocação do programa for realizada com o argumento *"-measure*", vão ser apresentados os tempos que o CPU demora a ordenar uma lista consoante o respetivo algoritmo de ordenação.

Inicialmente, define-se MAX_N como 10000000 (limite de tamanho de uma lista), N_MEASUREMENTS como 1000 (número máximo de análises para cada valor de n), N_EXTRA como 50 e MAX_TIME como 60 (máximo de segundos usados para cada valor de n). No entanto, como poderemos verificar mais tarde, iremos manipular esta constante com valores diferentes para analisar situações diferentes. O N_EXTRA serve para depois excluir os elementos das listas mais dispersos (menores e maiores valores) de forma a obter uma lista mais precisa.

Além das constantes, inicializam-se as variáveis *v, w e t* como *doubles*, sendo que *t* será um *array* de *doubles* com o tamanho do respetivo resultado aritmético. As variáveis *f_idx, n_idx, n, i* e *j* inicializam-se como inteiros e *data* é um ponteiro para uma estrutura de dados.

Na linha seguinte, alocamos memória para a lista de dados na variável *data*. Se esta tiver um valor nulo, imprimir-se-á uma mensagem de erro seguida de um encerramento do programa.

Utilizamos uma estrutura for loop para percorrer cada uma das *sorting functions* consoante o valor de f_*idx* (*index* da função). Começamos por imprimir o nome da função que

está a ser analisada e vamos mostrar para cada valor de n os tempos mínimos (*min time*), máximos (*max time*) e médios (*avg time*) do algoritmo, bem como o respetivo desvio padrão (*std dev*).

Usando outro for loop dentro do anterior para iterar o valor de *n_idx*, calculamos o valor de *n* e, caso este valor seja menor que MAX_N, executa-se uma série de passos.

Usamos o valor de *n_idx* para calcular uma *seed*, de forma a que todos os algoritmos de ordenação recebam os mesmos valores para o mesmo *n_idx*. De seguida, executa-se um for loop que vai de *i = 0* até à soma de N_MEASUREMENTS com o dobro do valor de N_EXTRA. Dentro deste, geramos até ao valor excluído do *index n* um número aleatório que vai ser armazenado em "*data*".

Ainda neste for loop, guardamos o valor de *cpu_time* na variável *v,* executamos a respetiva função de ordenação com os valores de entrada (*data, first = 0* e *n)* e, já concluída a disposição dos elementos, atualiza-se a variável *v* com o tempo que este último passo demorou a ser executado através da subtração do *cpu_time* atual com o antigo.

De seguida, usa-se o método *insertion sort* para introduzir o novo tempo de execução do algoritmo de ordenação no respetivo lugar do *array t,* que guarda todos os tempos de execução para um respetivo n.

Para calcular o tempo médio de execução, fazemos *v = 0.0*, somamos todos os valores do *array t* menos os últimos N_EXTRA elementos mais dispersos de cada extremidade, com tempos de execução maiores, e dividimos a soma por N_MEASUREMENTS.

Seguidamente, calcula-se o valor da variância com a soma do quadrado do valor de *t[i] – v* (média de tempo de execução) sobre o valor de N_MEASUREMENTS.

Tendo agora todos os dados fundamentais, prossegue-se para a apresentação do output. Mostra-se o valor de *min time*, o *max time*, o *avg time* (valor de v) e o desvio padrão, que é a raiz quadrada da variância (*sqrt(w)*). Usa-se a função *fflush* para garantir que todo o output do programa é, também, guardado num ficheiro de texto e, caso v * N_MEASUREMENTS for maior ou igual ao MAX_TIME, então estamos a passar muito tempo para esse valor de *n* e passamos para a próxima *sorting function* com o *break*.

3. Algoritmos de ordenação

3.1. Bubble Sort

O Bubble Sort é um algoritmo de ordenação bastante simples de ser implementado e de ser percebido, contudo comparado com os outros métodos de ordenação, o Bubble sort mostra-se como o algoritmo mais ineficiente de todo.

A ideia do Bubble Sort, é ir fazendo iterações começando no primeiro elemento e levar até para o final do array o maior valor presente no mesmo, depois na iteração seguinte meter o segundo maior na penúltima posição do array e assim sucessivamente.

Exemplo, simples , de como funciona o Bubble sort:

Array inicial é : [2, 4 ,6 , 3, 1 , 5].

1ª Iteração: [2, 4 , 3, 1, 5, 6] e ficamos a saber que já não é necessário verificar a última posição na próxima posição.

2ª Iteração : [2, 3 , 1 , 4 ,5 , 6], neste caso acontece uma particularidade que acaba por ser muito útil para otimizar o nosso tempo, que é se no nosso código utilizarmos uma variável chamada last, por exemplo, nesta variável vai ficar guardada a posição da última troca e assim na próxima sabemos que só precisamos de ordenar até essa posição que dessa para a frente já se encontra ordenado, neste caso last = 2 caso não se fizesse isto, iria ser preciso fazer uma iteração para meter o 5 na sua posição correta e outra para o 4 quando na realidade eles já se encontravam nessas posições e resolve-se com uma iteração em vez de duas.

3ª Iteração: [2 , 1, 3 ,4 ,5 ,6]

4ª Iteração: [1, 2, 3, 4, 5, 6]

5ª Iteração: o array não sofre nenhuma troca logo o algoritmo acaba.

Se verificar que não se realizou nenhuma troca, o algoritmo pode terminar automaticamente, pois significa que o array já esta ordenado.

Caso o array já esteja ordenado na 1ªIteração, vai corresponder ao melhor caso possível e tem complexidade computacional de O(n).

Contudo raramente um array já se encontra ordenado logo ao inicio, logo as complexidades computacionais que acaba por ser mais importante de se observar é a do caso médio onde a complexidade computacional é de O (n^2) e no pior caso de todos, que

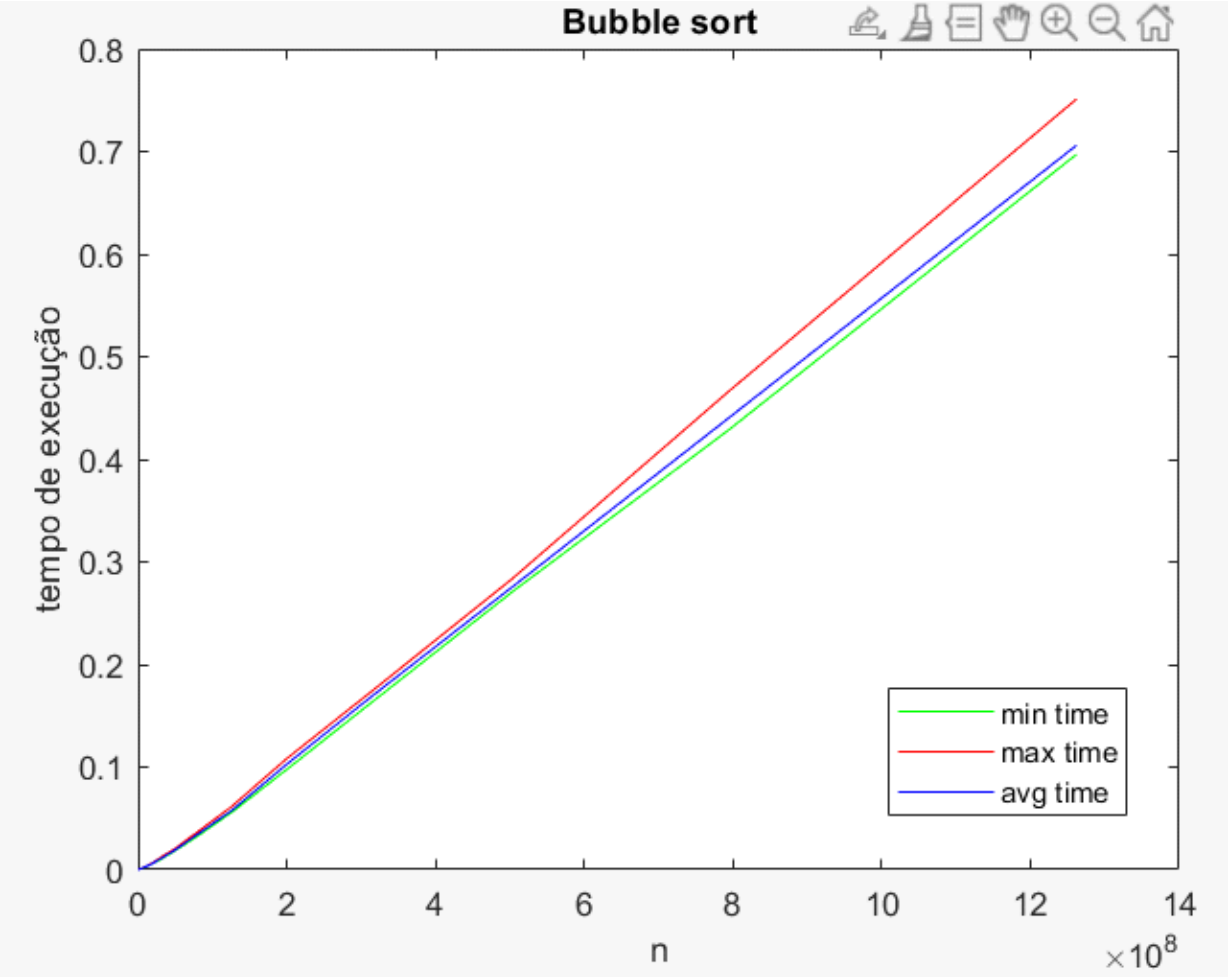
corresponde ao array estar ordenado por ordem inversa à partida a complexidade computacional é também de O(n^2).

No que toca a uma possível implementação em C pode-se implementar utilizando menos de 15 linhas, o que acaba por revelar que é um código que podemos considerar bastante compacto.

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	8.290e-07	9.580e-07	8.861e-07	3.011e-08
13	9.510e-07	1.112e-06	1.029e-06	3.945e-08
16	1.102e-06	1.377e-06	1.220e-06	5.938e-08
20	1.364e-06	1.967e-06	1.584e-06	1.436e-07
25	1.722e-06	2.241e-06	1.935e-06	1.076e-07
32	2.364e-06	3.079e-06	2.659e-06	1.535e-07
40	2.903e-06	3.938e-06	3.427e-06	2.644e-07
50	3.520e-06	5.212e-06	4.120e-06	3.421e-07
63	4.543e-06	5.838e-06	5.116e-06	3.265e-07
79	6.119e-06	7.595e-06	6.682e-06	2.950e-07
100	9.267e-06	1.281e-05	1.025e-05	7.315e-07
126	1.375e-05	1.826e-05	1.500e-05	6.709e-07
158	2.052e-05	2.492e-05	2.218e-05	8.516e-07
200	3.060e-05	3.695e-05	3.256e-05	1.111e-06
251	4.476e-05	5.655e-05	4.752e-05	1.781e-06
316	6.694e-05	7.756e-05	6.968e-05	1.708e-06
398	9.894e-05	1.226e-04	1.035e-04	3.480e-06
501	1.483e-04	1.975e-04	1.569e-04	9.741e-06
631	2.248e-04	2.893e-04	2.361e-04	1.176e-05
794	3.418e-04	4.462e-04	3.603e-04	1.875e-05
1000	5.284e-04	6.830e-04	5.565e-04	2.889e-05
1259	8.304e-04	1.048e-03	8.712e-04	4.165e-05
1585	1.301e-03	1.682e-03	1.377e-03	7.403e-05
1995	2.113e-03	2.645e-03	2.241e-03	1.244e-04
2512	3.464e-03	4.235e-03	3.704e-03	1.973e-04
3162	5.903e-03	7.051e-03	6.312e-03	2.724e-04
3981	1.052e-02	1.251e-02	1.127e-02	4.603e-04
5012	1.859e-02	2.095e-02	1.944e-02	5.588e-04
6310	3.254e-02	3.661e-02	3.426e-02	9.376e-04
7943	5.602e-02	6.180e-02	5.785e-02	1.330e-03
10000	9.769e-02	1.083e-01	1.028e-01	2.441e-03
12589	1.647e-01	1.755e-01	1.705e-01	2.532e-03
15849	2.703e-01	2.831e-01	2.751e-01	2.168e-03
19953	4.298e-01	4.674e-01	4.414e-01	9.595e-03
25119	6.972e-01	7.511e-01	7.060e-01	1.114e-02

Gráfico dos valores obtidos:



3.2. Shaker sort

O algoritmo de ordenação Shaker sort é um algoritmo com filosofia semelhante ao do Bubble Sort, só que este algoritmo coloca o elemento mais pequeno na primeira posição possível, ou seja, no Bubble sort colocamos, ao fim de cada iteração, o maior elemento no fim, no Shaker sort ao fim de cada iteração colocamos o maior elemento no fim e ainda o elemento mais pequeno no início. Logo, este algoritmo acaba por ser uma melhoria em relação ao Bubble Sort.

Exemplo, simples, de como funciona o Shaker sort:

Array inicial é : [2, 4 ,6 , 3, 1 , 5].

1ªIteração:

Up Pass

[2, 4 ,3 ,1, 5 , 6].

Down Pass:

[1, 2, 4, 3, 5, 6]

2ªIteração

Up Pass

[1, 2, 3, 4, 5, 6].

Down Pass:

[1, 2, 3, 4, 5, 6] => não se verifica nenhuma troca e acaba o algoritmo.

De salientar que, neste algoritmo de ordenação, podemos utilizar a mesma estratégia que havia sido utilizada para o Bubble sort, ou seja, criar uma variável com o nome last, onde vai ficar colocada o endereço da última troca. Contudo, neste algoritmo, pensado no conjunto como um chão, primeiro elemento, e um teto, último elemento, no bubble sort apenas baixamos o teto, neste algoritmo podemos baixar o teto e subir o chão, pois sabemos que os elementos que deixaram de entrar no novo intervalo já estão colocados da forma correta.

Ao nível de complexidades computacionais, o melhor caso é o mesmo que o Bubble sort, logo, O(n).

O caso médio e o pior caso, este é o mesmo que o buble sort também, a complexidade computacional são de O(n^2).

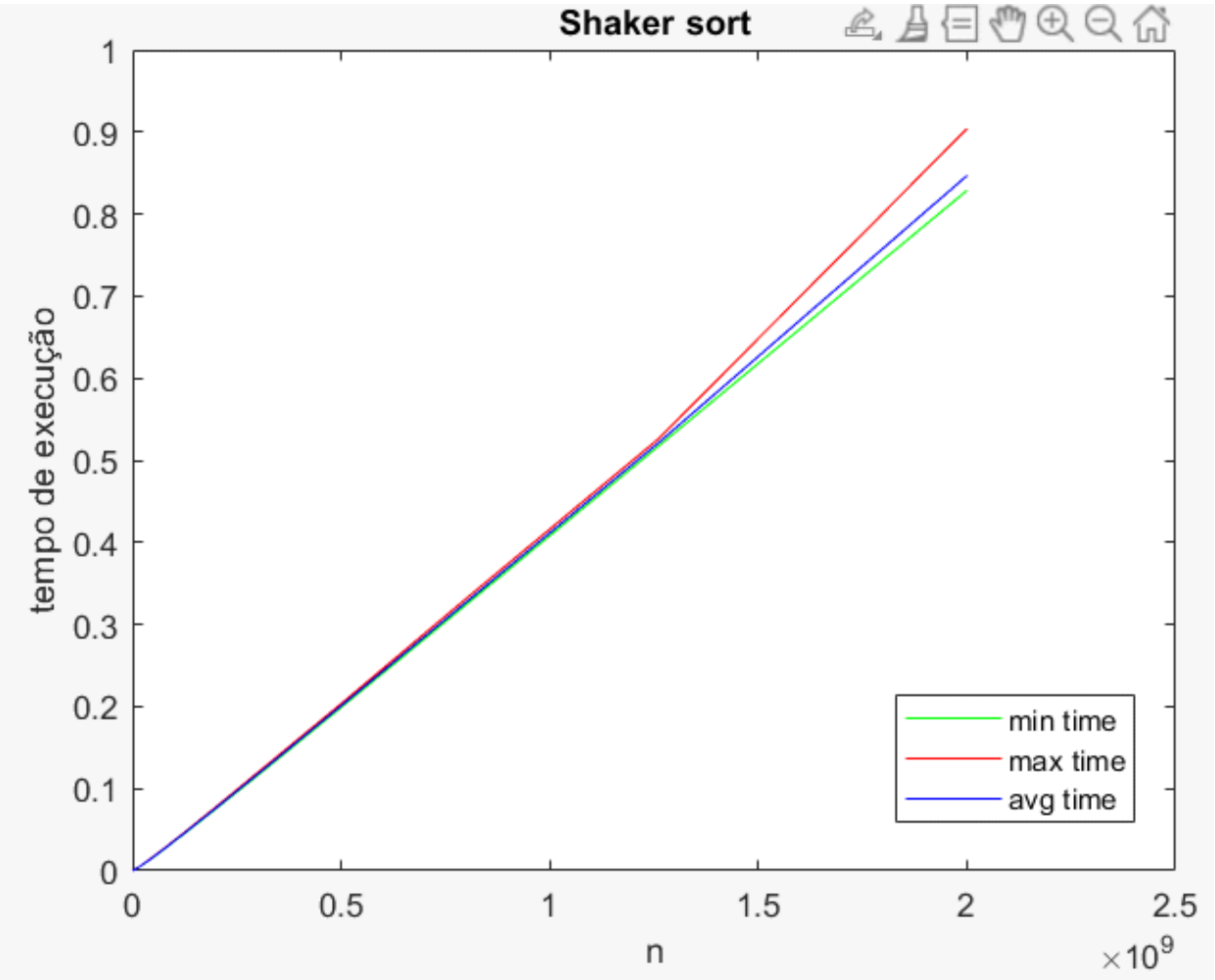
Contudo, é importante salientar que, no caso médio, a constante que se encontra antes do n^2 é inferior no Shaker sort em relação ao Bubble sort. Podemos verificar isso pelos tempos médios de execução que foram obtidos e que se encontram presentes na tabela de valores, tanto que, para o intervalo de tempo de 600 segundos, o Shaker Sort conseguiu fazer para um array de tamanho 31623 e o Bubble Sort, neste intervalo de tempo, ficou no valor anterior, 25119.

Contudo, este código acaba por ser menos compacto do que o código do Bubble sort, o que é uma desvantagem. Isto pelo facto de ser necessário adicionar um for para fazer o Down Pass.

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	5.240e-07	6.110e-07	5.641e-07	2.009e-08
13	5.900e-07	6.930e-07	6.398e-07	2.411e-08
16	6.700e-07	7.990e-07	7.317e-07	2.979e-08
20	7.950e-07	9.650e-07	8.787e-07	4.146e-08
25	9.820e-07	1.206e-06	1.095e-06	5.382e-08
32	1.320e-06	1.629e-06	1.472e-06	7.410e-08
40	1.782e-06	2.387e-06	2.035e-06	1.378e-07
50	2.449e-06	3.029e-06	2.745e-06	1.349e-07
63	3.555e-06	4.381e-06	3.968e-06	1.949e-07
79	5.267e-06	6.995e-06	5.954e-06	4.274e-07
100	7.809e-06	9.253e-06	8.484e-06	3.398e-07
126	1.165e-05	1.367e-05	1.264e-05	4.765e-07
158	1.740e-05	2.017e-05	1.877e-05	6.609e-07
200	2.642e-05	3.091e-05	2.840e-05	9.947e-07
251	3.992e-05	4.514e-05	4.240e-05	1.244e-06
316	6.024e-05	6.682e-05	6.351e-05	1.519e-06
398	9.174e-05	1.001e-04	9.559e-05	1.994e-06
501	1.392e-04	1.510e-04	1.452e-04	2.727e-06
631	2.139e-04	2.288e-04	2.213e-04	3.653e-06
794	3.295e-04	3.505e-04	3.397e-04	5.027e-06
1000	5.128e-04	5.409e-04	5.267e-04	6.808e-06
1259	8.025e-04	8.434e-04	8.237e-04	9.768e-06
1585	1.280e-03	1.344e-03	1.311e-03	1.486e-05
1995	2.077e-03	2.206e-03	2.130e-03	2.834e-05
2512	3.434e-03	3.597e-03	3.515e-03	3.904e-05
3162	5.778e-03	6.064e-03	5.904e-03	6.522e-05
3981	9.783e-03	1.050e-02	1.001e-02	1.336e-04
5012	1.650e-02	1.717e-02	1.682e-02	1.599e-04
6310	2.766e-02	2.867e-02	2.810e-02	2.295e-04
7943	4.582e-02	4.736e-02	4.649e-02	3.459e-04
10000	7.536e-02	7.776e-02	7.636e-02	5.296e-04
12589	1.230e-01	1.267e-01	1.246e-01	7.844e-04
15849	1.998e-01	2.044e-01	2.019e-01	1.063e-03
19953	3.227e-01	3.301e-01	3.257e-01	1.606e-03
25119	5.178e-01	5.267e-01	5.219e-01	2.008e-03
31623	8.288e-01	9.040e-01	8.468e-01	1.946e-02

Gráfico dos valores obtidos:



3.3. Insertion sort

Outro algoritmo de ordenação é o Insertion sort, e este em regra geral, é comparado ao algoritmo do Bubble e do Shaker sort, visto que este possui as mesmas complexidades computacionais que os outros 2 referidos anteriormente. Contudo este método acaba por ser mais rápido pois as suas constantes de multiplicação são mais pequenas e também porque este método escreve muito menos coisas na memória e assim também vai ler muito menos coisas na memória.

Tanto que no intervalo de tempo referido, 600 segundos, o tamanho máximo que o array havia conseguido alcançar era de 31623 elementos e no insertion sort chegou aos 79433 elementos, logo conseguiu ordenar neste intervalo vetores com o dobro do tamanho do máximo alcançado pelo Shaker.

Explicando o conceito do Insertion sort, é um método de ordenação muito simples, consiste em começar no segundo elemento do array e compara-lo com o primeiro elemento, se o segundo for menor que o primeiro troca-se um com o outro, depois vamos para o terceiro elemento, comparamos com o segundo se for menor, troca-se, caso seja maior não trocamos e já não vai ser preciso comparar com o primeiro, ou seja, podemos passar logo ao próximo elemento do array. O conceito base é agarrar no elemento, deixando um buraco no array depois os elementos maiores vão ser deslocados para a direita, deixando o buraco o mais à esquerda possível e no final o elemento que havia sido agarrado no início vai tapar esse buraco.

Exemplo, simples, de como funciona o Insertion sort:

Array inicial é: [2, 4 ,6 , 3, 1 , 5].

1ªIteração

Verificar se 4< 2? Trocar e verificar outra vez : fica onde está

[2, 4, 6, 3, 1 ,5]

2ªIteração

Verificar se 6< 4? Trocar e verificar outra vez: fica onde está

[2, 4, 6, 3, 1 ,5]

3ªIteração

Verificar se 3< 6? Trocar e verificar outra vez : fica onde está

Verificar se 3<4? Trocar e verificar outra vez : fica onde está

Verificar se 3<2? Trocar e verificar outra vez : fica onde está

[2, 3 , 4, 6, 1 ,5]

4ªIteração

Verificar se 1< 6? Trocar e verificar outra vez : fica onde está

Verificar se 1<4? Trocar e verificar outra vez : fica onde está

Verificar se 1<3? Trocar e verificar outra vez : fica onde está

Verificar se 1<2? Trocar e verificar outra vez : fica onde está

[1, 2, 3 , 4, 6 ,5]

5ªIteração

Verificar se 5< 6? Trocar e verificar outra vez : fica onde está

Verificar se 5<4? Trocar e verificar outra vez: fica onde está

[1, 2, 3 , 4, 5, 6]

E como chegamos ao último elemento do array terminamos o código.

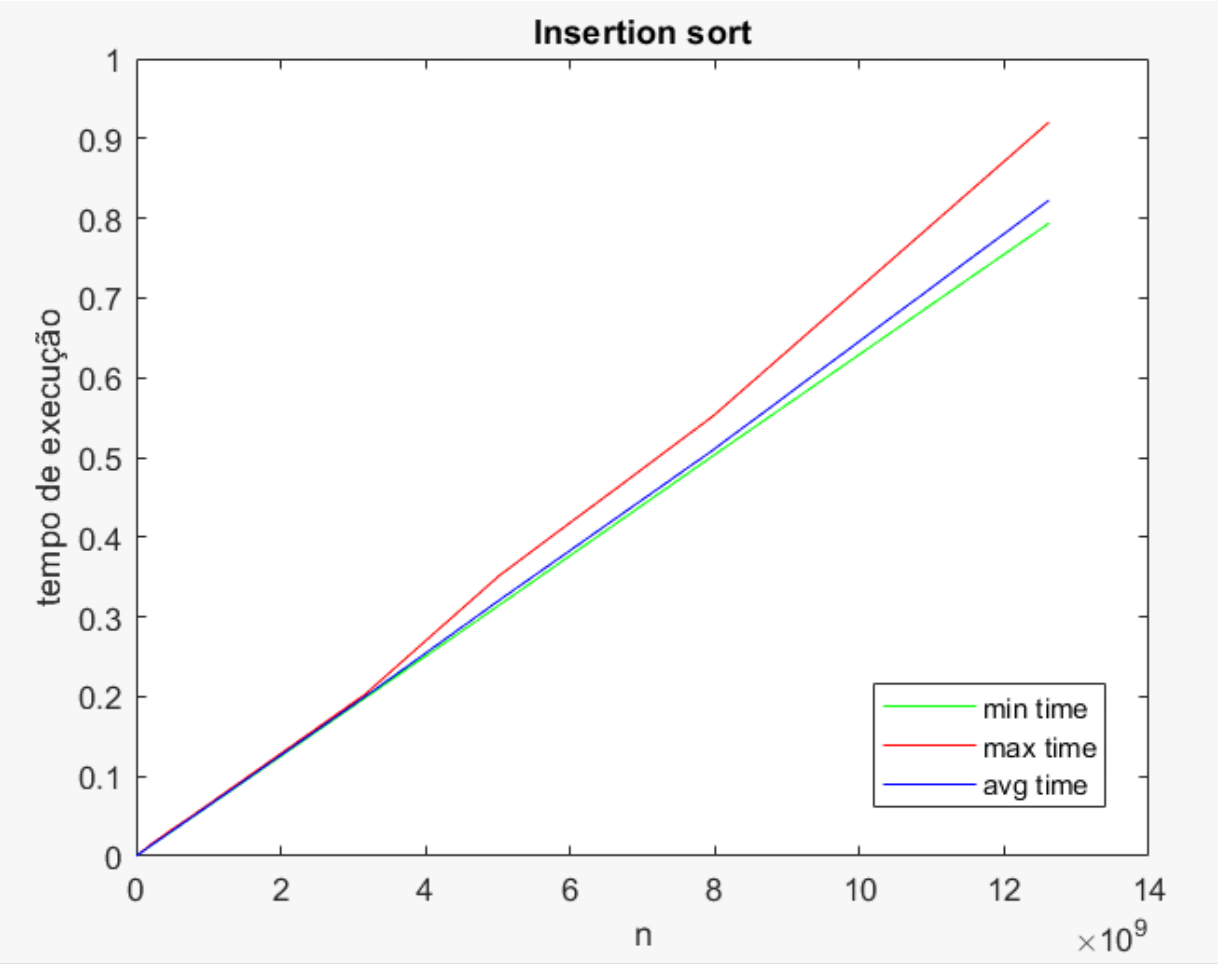
O código do Insertion sort é de todos os métodos de ordenação o mais compacto de todos, pois podemos implementar este método de ordenação utilizando menos de 10 linhas de código em C.

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	5.040e-07	5.410e-07	5.209e-07	8.627e-09
13	5.370e-07	5.760e-07	5.554e-07	9.157e-09
16	5.740e-07	8.140e-07	6.548e-07	6.864e-08
20	6.180e-07	6.790e-07	6.425e-07	1.381e-08
25	6.810e-07	1.018e-06	7.394e-07	6.584e-08
32	7.830e-07	1.874e-06	9.349e-07	2.242e-07
40	9.100e-07	1.498e-06	1.053e-06	1.379e-07
50	1.084e-06	1.228e-06	1.150e-06	3.533e-08
63	1.349e-06	1.586e-06	1.446e-06	5.153e-08
79	1.754e-06	2.727e-06	1.990e-06	2.557e-07
100	2.366e-06	3.353e-06	2.604e-06	1.593e-07
126	3.292e-06	3.950e-06	3.558e-06	1.396e-07
158	4.631e-06	7.332e-06	5.076e-06	4.556e-07
200	6.774e-06	8.190e-06	7.300e-06	2.864e-07
251	1.003e-05	1.685e-05	1.105e-05	1.392e-06
316	1.504e-05	2.454e-05	1.628e-05	1.613e-06
398	2.278e-05	3.340e-05	2.432e-05	1.330e-06
501	3.506e-05	4.425e-05	3.723e-05	1.417e-06
631	5.431e-05	6.564e-05	5.711e-05	1.781e-06
794	8.440e-05	9.952e-05	8.870e-05	2.783e-06
1000	1.299e-04	1.435e-04	1.355e-04	2.981e-06
1259	2.039e-04	3.825e-04	2.264e-04	3.397e-05
1585	3.243e-04	5.436e-04	3.554e-04	4.232e-05
1995	5.029e-04	6.342e-04	5.320e-04	2.691e-05
2512	7.923e-04	8.546e-04	8.146e-04	1.317e-05
3162	1.253e-03	1.323e-03	1.279e-03	1.484e-05
3981	1.982e-03	2.154e-03	2.027e-03	2.867e-05
5012	3.152e-03	3.669e-03	3.281e-03	1.052e-04
6310	5.008e-03	5.872e-03	5.197e-03	1.589e-04
7943	7.951e-03	8.838e-03	8.194e-03	1.586e-04
10000	1.252e-02	1.456e-02	1.295e-02	4.034e-04
12589	1.985e-02	2.202e-02	2.031e-02	4.550e-04
15849	3.142e-02	3.397e-02	3.196e-02	5.493e-04

19953	4.977e-02	5.186e-02	5.034e-02	4.184e-04
25119	7.891e-02	8.164e-02	7.975e-02	5.999e-04
31623	1.252e-01	1.287e-01	1.265e-01	8.496e-04
39811	1.985e-01	2.033e-01	2.003e-01	1.108e-03
50119	3.150e-01	3.521e-01	3.212e-01	7.317e-03
63096	5.013e-01	5.509e-01	5.085e-01	8.857e-03
79433	7.938e-01	9.206e-01	8.225e-01	3.071e-02

Gráfico dos valores obtidos:



3.4. Shell Sort

Este algoritmo de ordenação é uma modificação do insertion sort, ou seja, consiste em aplicar sucessivamente o algoritmo do insertion sort a subarrays do array base.

Basicamente o array a[0], a[1], …. , a[n-1] é subdivido em sub array's :

a[0], a[h+0], a[2h+0], ..

a[1], a[h+1], a[2h+1], ..

a[h-1], a[2h-1], a[3h-1], ..

O h, stride, é possível utilizar-se qualquer sequência desde que o último stride seja de valor 1, logo, isto implica que consoante o h que seja escolhido, as complexidades computacionais podem tornar-se melhores ou iguais, que o Insertion sort no caso médio, ou seja, O(n^2).

Na implementação deste código, o caso médio ficou com uma complexidade computacional de O(n^4/3). Podemos verificar isto pois, no tempo de execução que foi escolhido, 600 segundos, os valores que obtivemos, chegou a um array máximo de tamanho 5011872 enquanto que no insertion sort, o tamanho máximo de array que tinha sido possível ordenar era 79433.

Isto leva-nos a concluir que o stride escolhido foi bastante bom, pois conseguiu ordenar valores muito maiores do que no insertion sort. Como já foi explicado o shell sort é uma modificação do insertion sort.

No que toca a implementação de código, o código continua a ser um código bastante curto, ou seja, podemos considerar um código compacto, mas não é tão compacto quanto o código do insertion sort.

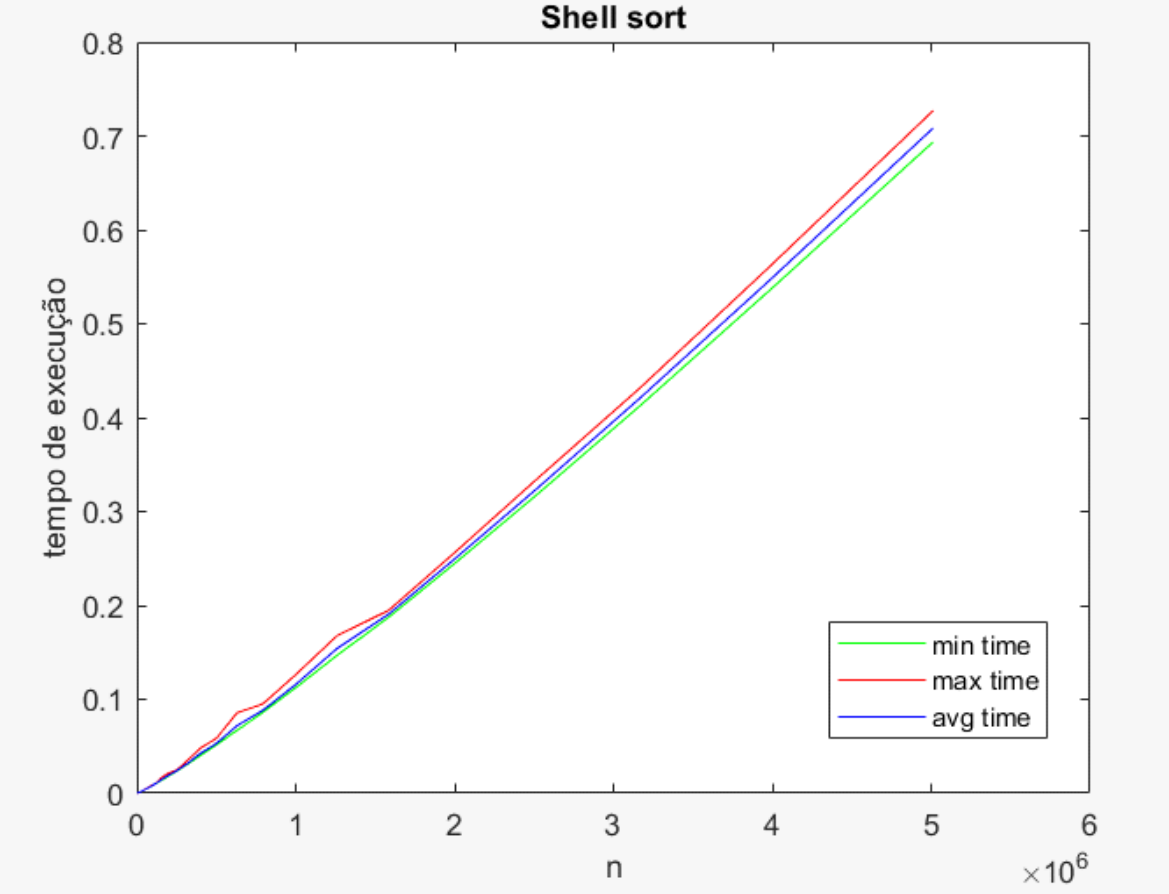
Outra desvantagem é também o melhor caso que passa de O(n) para O(n.log(n)).

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	5.190e-07	5.910e-07	5.509e-07	1.569e-08
13	5.690e-07	6.470e-07	6.062e-07	1.797e-08
16	6.260e-07	7.130e-07	6.689e-07	1.987e-08
20	7.140e-07	8.190e-07	7.657e-07	2.468e-08
25	8.230e-07	9.560e-07	8.839e-07	3.008e-08
32	9.770e-07	1.113e-06	1.044e-06	3.287e-08
40	1.153e-06	1.323e-06	1.235e-06	3.874e-08
50	1.446e-06	1.643e-06	1.547e-06	4.787e-08
63	1.808e-06	2.046e-06	1.924e-06	5.544e-08
79	2.273e-06	2.534e-06	2.399e-06	6.382e-08
100	2.924e-06	3.239e-06	3.079e-06	7.532e-08
126	3.802e-06	4.146e-06	3.975e-06	8.334e-08
158	5.009e-06	5.453e-06	5.218e-06	1.023e-07
200	6.562e-06	7.051e-06	6.809e-06	1.173e-07
251	8.473e-06	9.037e-06	8.765e-06	1.355e-07

316	1.106e-05	1.168e-05	1.138e-05	1.492e-07
398	1.455e-05	1.533e-05	1.495e-05	1.854e-07
501	1.918e-05	2.006e-05	1.960e-05	2.017e-07
631	2.512e-05	2.730e-05	2.564e-05	2.912e-07
794	3.275e-05	3.450e-05	3.339e-05	3.253e-07
1000	4.282e-05	4.420e-05	4.347e-05	3.347e-07
1259	5.618e-05	5.809e-05	5.708e-05	4.436e-07
1585	7.391e-05	7.958e-05	7.499e-05	7.330e-07
1995	9.626e-05	9.886e-05	9.740e-05	5.921e-07
2512	1.256e-04	1.295e-04	1.271e-04	8.302e-07
3162	1.632e-04	1.685e-04	1.651e-04	1.086e-06
3981	2.136e-04	2.239e-04	2.162e-04	1.543e-06
5012	2.790e-04	2.871e-04	2.822e-04	1.656e-06
6310	3.623e-04	3.868e-04	3.668e-04	3.291e-06
7943	4.710e-04	4.904e-04	4.764e-04	3.642e-06
10000	6.120e-04	6.402e-04	6.192e-04	5.499e-06
12589	7.991e-04	8.254e-04	8.065e-04	4.416e-06
15849	1.039e-03	1.088e-03	1.050e-03	7.498e-06
19953	1.347e-03	1.409e-03	1.362e-03	1.201e-05
25119	1.750e-03	1.846e-03	1.773e-03	1.808e-05
31623	2.270e-03	2.373e-03	2.299e-03	2.175e-05
39811	2.948e-03	3.081e-03	2.984e-03	2.613e-05
50119	3.832e-03	4.080e-03	3.885e-03	4.302e-05
63096	4.968e-03	5.261e-03	5.036e-03	5.371e-05
79433	6.437e-03	6.893e-03	6.542e-03	8.921e-05
100000	8.342e-03	8.862e-03	8.466e-03	9.042e-05
125893	1.084e-02	1.149e-02	1.101e-02	1.347e-04
158489	1.411e-02	1.717e-02	1.493e-02	6.729e-04
199526	1.823e-02	2.165e-02	1.914e-02	8.750e-04
251189	2.354e-02	2.468e-02	2.396e-02	2.655e-04
316228	3.058e-02	3.459e-02	3.140e-02	7.181e-04
398107	3.983e-02	4.774e-02	4.210e-02	1.748e-03
501187	5.150e-02	5.859e-02	5.309e-02	1.364e-03
630957	6.708e-02	8.565e-02	7.201e-02	4.500e-03
794328	8.623e-02	9.526e-02	8.846e-02	1.771e-03
1000000	1.124e-01	1.263e-01	1.158e-01	3.094e-03
1258925	1.467e-01	1.676e-01	1.540e-01	5.346e-03
1584893	1.874e-01	1.948e-01	1.907e-01	1.729e-03
1995262	2.440e-01	2.555e-01	2.488e-01	2.693e-03
2511886	3.171e-01	3.337e-01	3.235e-01	3.634e-03
3162278	4.119e-01	4.309e-01	4.201e-01	4.519e-03
3981072	5.353e-01	5.605e-01	5.460e-01	5.978e-03
5011872	6.935e-01	7.269e-01	7.083e-01	7.866e-03

Gráfico dos valores obtidos:



3.5. Quick Sort

O Quick sort é um algoritmo de ordenação de informação que segue a filosofia *divide and conquer*, isto é, transforma o problema em sub-problemas de forma a tornar o processo de ordenação mais simples.

Inicialmente, escolhe-se um elemento aleatório do array para ser o pivot. Este elemento selecionado, que neste caso vai ser sempre o último elemento da lista, vai servir como item de comparação com os restantes elementos da lista.

De seguida, verificamos para cada elemento da lista se são superiores, iguais ou inferiores ao valor estabelecido como pivot. Caso o elemento for menor que o pivot, este é colocado num array que vai juntar todos os valores menores que este último. Se o item for igual ao pivot, este é colocado num array com valores iguais ao pivot. Por último, se o elemento for menor que o pivot, este vai ser adicionado ao array que armazena todos os valores menores que o pivot.

Agora que temos a lista inicial dividida em sublistas, repetimos o mesmo processo para as listas menores e maiores que o pivot inicial, uma vez que não precisamos de ordenar a lista com valores iguais

ao pivot. Desta forma, o algoritmo quick sort é uma função recursiva que vai ordenando as sublistas do array original e, no final, temos todos os elementos da lista original na respetiva posição correta.

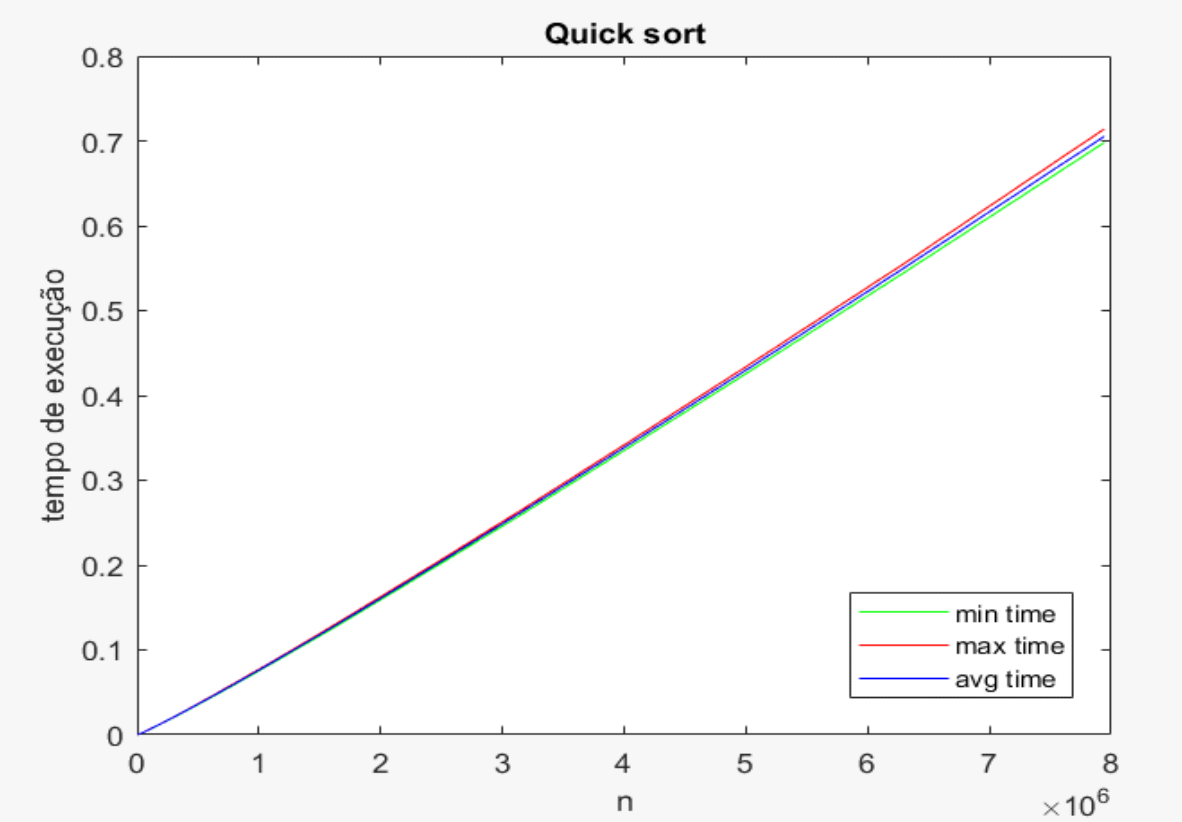
Quanto à complexidade, a eficiência do algoritmo depende da escolha do pivot. Se o array já está ordenado de forma crescente, decrescente ou se todos os elementos são iguais, vai ocorrer o pior caso do algoritmo – $O(n^2)$. No caso contrário, o melhor caso do quick sort ocorre quando cada pivot da lista é selecionado de forma a criar sublistas do mesmo tamanho, ou seja, equilibradas – O(nlog(n)). Normalmente, o quick sort tem uma complexidade de O(nlog(n)).

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	4.960e-07	5.330e-07	5.131e-07	8.476e-09
13	5.260e-07	5.660e-07	5.451e-07	9.528e-09
16	5.610e-07	6.080e-07	5.816e-07	1.090e-08
20	6.460e-07	7.580e-07	6.863e-07	2.060e-08
25	7.210e-07	8.250e-07	7.666e-07	2.466e-08
32	8.380e-07	9.800e-07	9.053e-07	3.362e-08
40	1.022e-06	1.356e-06	1.117e-06	7.389e-08
50	1.237e-06	1.597e-06	1.340e-06	7.002e-08
63	1.522e-06	1.734e-06	1.627e-06	5.089e-08
79	1.923e-06	2.177e-06	2.046e-06	6.181e-08
100	2.477e-06	2.800e-06	2.629e-06	7.385e-08
126	3.209e-06	3.565e-06	3.385e-06	8.279e-08
158	4.147e-06	4.574e-06	4.352e-06	9.887e-08
200	5.419e-06	5.909e-06	5.658e-06	1.149e-07
251	7.017e-06	7.638e-06	7.320e-06	1.405e-07
316	9.145e-06	9.868e-06	9.525e-06	1.686e-07
398	1.196e-05	1.285e-05	1.240e-05	2.100e-07
501	1.566e-05	1.672e-05	1.619e-05	2.479e-07
631	2.045e-05	2.169e-05	2.110e-05	3.002e-07
794	2.672e-05	2.836e-05	2.754e-05	3.783e-07
1000	3.495e-05	3.683e-05	3.590e-05	4.360e-07
1259	4.550e-05	4.803e-05	4.680e-05	5.755e-07
1585	5.934e-05	6.250e-05	6.096e-05	7.194e-07
1995	7.729e-05	8.121e-05	7.931e-05	9.078e-07
2512	1.008e-04	1.056e-04	1.032e-04	1.097e-06
3162	1.313e-04	1.370e-04	1.341e-04	1.352e-06
3981	1.703e-04	1.790e-04	1.743e-04	1.814e-06
5012	2.213e-04	2.301e-04	2.258e-04	2.163e-06
6310	2.869e-04	2.983e-04	2.929e-04	2.600e-06
7943	3.714e-04	3.878e-04	3.793e-04	3.563e-06
10000	4.821e-04	5.001e-04	4.910e-04	4.287e-06
12589	6.244e-04	6.473e-04	6.355e-04	5.151e-06
15849	8.056e-04	8.348e-04	8.204e-04	6.969e-06

19953	1.042e-03	1.078e-03	1.060e-03	8.707e-06
25119	1.346e-03	1.390e-03	1.369e-03	1.031e-05
31623	1.734e-03	1.817e-03	1.768e-03	1.717e-05
39811	2.240e-03	2.337e-03	2.279e-03	2.131e-05
50119	2.887e-03	3.014e-03	2.940e-03	2.705e-05
63096	3.725e-03	3.869e-03	3.786e-03	3.079e-05
79433	4.789e-03	4.945e-03	4.869e-03	3.557e-05
100000	6.172e-03	6.362e-03	6.267e-03	4.587e-05
125893	7.936e-03	8.168e-03	8.054e-03	5.370e-05
158489	1.021e-02	1.049e-02	1.035e-02	6.888e-05
199526	1.312e-02	1.350e-02	1.331e-02	8.636e-05
251189	1.687e-02	1.735e-02	1.712e-02	1.073e-04
316228	2.166e-02	2.223e-02	2.196e-02	1.374e-04
398107	2.781e-02	2.857e-02	2.819e-02	1.732e-04
501187	3.569e-02	3.665e-02	3.619e-02	2.219e-04
630957	4.577e-02	4.702e-02	4.640e-02	2.871e-04
794328	5.868e-02	6.033e-02	5.950e-02	3.593e-04
1000000	7.530e-02	7.714e-02	7.625e-02	4.329e-04
1258925	9.657e-02	9.890e-02	9.775e-02	5.391e-04
1584893	1.236e-01	1.266e-01	1.252e-01	6.900e-04
1995262	1.587e-01	1.623e-01	1.605e-01	8.530e-04
2511886	2.034e-01	2.076e-01	2.056e-01	9.909e-04
3162278	2.601e-01	2.656e-01	2.631e-01	1.291e-03
3981072	3.334e-01	3.399e-01	3.368e-01	1.564e-03
5011872	4.267e-01	4.349e-01	4.309e-01	2.025e-03
6309573	5.458e-01	5.565e-01	5.513e-01	2.609e-03
7943282	6.981e-01	7.141e-01	7.054e-01	3.443e-03

Gráfico dos valores obtidos:



3.6. Merge Sort

O Merge sort é outro algoritmo de ordenação indêntico ao Quick sort. Utiliza a mesma filosofia deste último, contudo enquanto o Quick sort utiliza um pivot aleatório na divisão do array, o Merge divide o array a metade, isto é, o seu pivot será sempre o meio do array.

Após a divisão, se cada parte tiver um comprimento (one_after_last - first) maior que 40, recorre-se à recursividade para voltar a dividir aquela parte a metade. Quando cada parte for menor que 40 em comprimento, essas mesmas partes irão ser ordenadas recorrendo ao Insertion sort.

Este algoritmo não é tão bom quanto o Quick sort porque requere espaço de memória extra. Porém, o código do Merge sort é mais compacto, pois podemos implementar este método de ordenação utilizando menos de metade das linhas de código que do Quick sort.

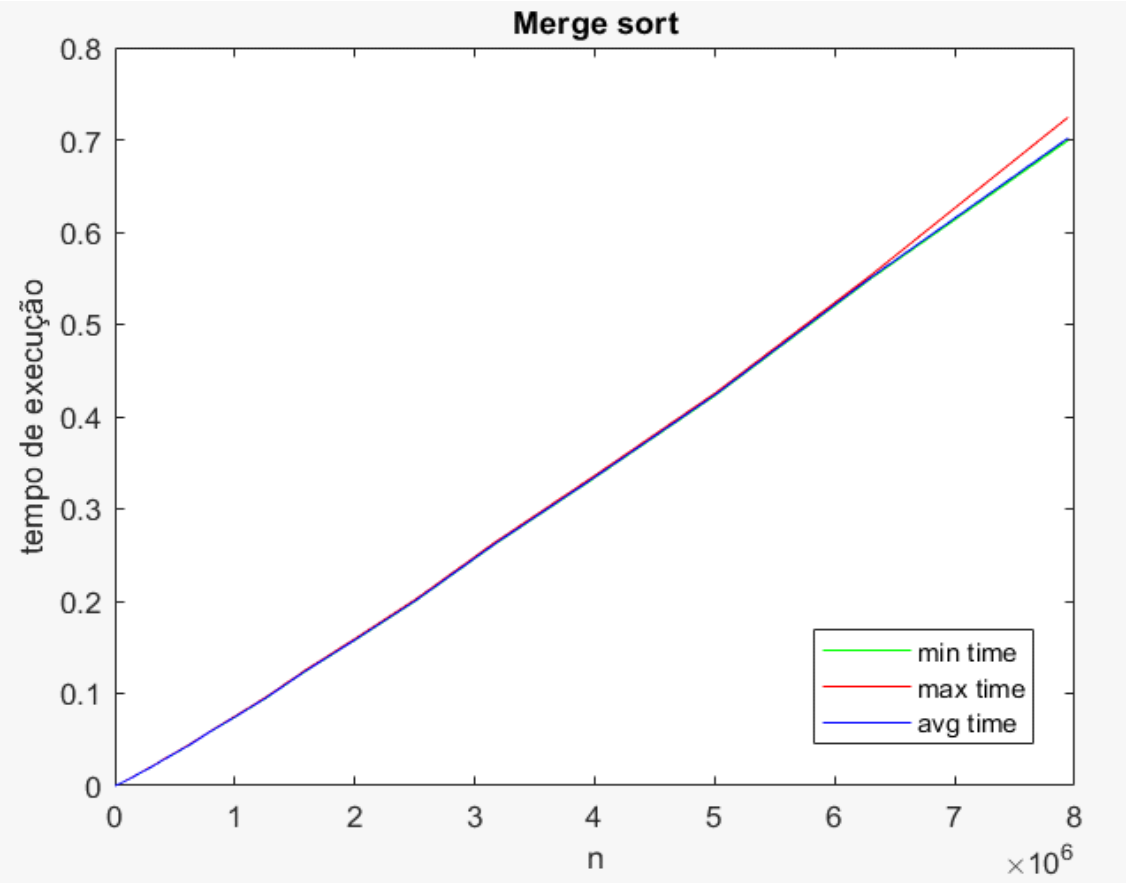
Em relação à sua complexidade computacional, tanto no pior e melhor caso, como no médio, será O(n.log(n)).

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	4.980e-07	5.350e-07	5.162e-07	8.674e-09
13	5.250e-07	5.710e-07	5.450e-07	1.040e-08
16	5.580e-07	6.050e-07	5.790e-07	1.117e-08
20	6.030e-07	7.240e-07	6.301e-07	1.816e-08
25	6.640e-07	7.330e-07	6.928e-07	1.648e-08
32	7.610e-07	8.500e-07	8.004e-07	2.143e-08
40	9.620e-07	1.061e-06	1.008e-06	2.342e-08
50	1.133e-06	1.258e-06	1.190e-06	2.977e-08
63	1.361e-06	1.503e-06	1.428e-06	3.379e-08
79	1.764e-06	1.942e-06	1.843e-06	4.186e-08
100	2.279e-06	2.498e-06	2.378e-06	4.987e-08
126	2.879e-06	3.150e-06	3.003e-06	6.358e-08
158	3.822e-06	4.115e-06	3.961e-06	6.935e-08
200	5.043e-06	5.382e-06	5.192e-06	7.816e-08
251	6.411e-06	6.812e-06	6.591e-06	9.322e-08
316	8.626e-06	9.124e-06	8.847e-06	1.158e-07
398	1.133e-05	1.185e-05	1.159e-05	1.238e-07
501	1.449e-05	1.510e-05	1.479e-05	1.456e-07
631	1.937e-05	2.012e-05	1.971e-05	1.751e-07
794	2.551e-05	2.632e-05	2.591e-05	1.888e-07
1000	3.272e-05	3.374e-05	3.318e-05	2.315e-07
1259	4.326e-05	4.451e-05	4.382e-05	2.793e-07
1585	5.717e-05	5.900e-05	5.782e-05	3.456e-07
1995	7.317e-05	7.487e-05	7.391e-05	3.832e-07
2512	9.594e-05	9.812e-05	9.684e-05	4.822e-07
3162	1.268e-04	1.296e-04	1.279e-04	5.438e-07
3981	1.624e-04	1.656e-04	1.635e-04	6.537e-07
5012	2.112e-04	2.155e-04	2.126e-04	8.158e-07
6310	2.794e-04	2.854e-04	2.810e-04	1.023e-06
7943	3.578e-04	3.631e-04	3.595e-04	1.043e-06
10000	4.605e-04	4.678e-04	4.628e-04	1.451e-06
12589	6.095e-04	6.188e-04	6.121e-04	1.614e-06
15849	7.774e-04	7.911e-04	7.807e-04	2.115e-06
19953	9.983e-04	1.013e-03	1.002e-03	2.656e-06
25119	1.320e-03	1.365e-03	1.325e-03	5.483e-06
31623	1.684e-03	1.735e-03	1.692e-03	1.217e-05
39811	2.158e-03	2.212e-03	2.170e-03	1.461e-05
50119	2.844e-03	2.898e-03	2.855e-03	1.386e-05
63096	3.622e-03	3.699e-03	3.636e-03	1.707e-05
79433	4.639e-03	4.695e-03	4.649e-03	8.918e-06
100000	6.093e-03	6.190e-03	6.111e-03	2.134e-05
125893	7.751e-03	7.850e-03	7.777e-03	2.475e-05
158489	9.917e-03	1.006e-02	9.966e-03	3.640e-05

199526	1.300e-02	1.321e-02	1.304e-02	3.031e-05
251189	1.656e-02	1.683e-02	1.660e-02	3.627e-05
316228	2.112e-02	2.138e-02	2.116e-02	3.951e-05
398107	2.759e-02	2.805e-02	2.766e-02	5.909e-05
501187	3.506e-02	3.551e-02	3.516e-02	7.490e-05
630957	4.480e-02	4.550e-02	4.490e-02	9.757e-05
794328	5.847e-02	5.899e-02	5.860e-02	9.275e-05
1000000	7.430e-02	7.520e-02	7.448e-02	1.771e-04
1258925	9.493e-02	9.594e-02	9.516e-02	1.979e-04
1584893	1.237e-01	1.250e-01	1.240e-01	2.493e-04
1995262	1.573e-01	1.588e-01	1.576e-01	2.900e-04
2511886	2.008e-01	2.025e-01	2.012e-01	3.700e-04
3162278	2.612e-01	2.633e-01	2.618e-01	4.377e-04
3981072	3.321e-01	3.345e-01	3.328e-01	5.480e-04
5011872	4.237e-01	4.263e-01	4.246e-01	6.406e-04
6309573	5.504e-01	5.541e-01	5.516e-01	7.607e-04
7943282	6.993e-01	7.243e-01	7.018e-01	3.941e-03

Gráfico dos valores obtidos:



3.7. Heap Sort

Antes de explicar como funciona o Heap Sort, é necessário perceber o que é um max-heap. Um max-heap é, praticamente, uma árvore binária em que cada elemento-pai é sempre maior ou igual aos elementos-filho. Assim, temos uma lista em que o item no index *i* é menor ou igual que aquele situado em *floor*(*i*/2).

Para ordenar uma lista, o algoritmo heap sort vai, inicialmente, transformar o array original (heap) num max-heap. Desta forma, segue-se o critério apresentado no parágrafo anterior. Concluída a criação do max-heap, o elemento que está no topo será o maior entre todos os restantes e, uma vez que queremos armazenar este elemento no fim da lista, substituímos este nó (index 0) pelo último nó, passando este último a estar no topo do max-heap. No final, removemos o nó situado na última posição do array, que era o valor máximo do max-heap.

De seguida, como os elementos do max-heap anterior foram trocados, este terá que ser construído novamente, prosseguindo-se aos mesmos passos realizados anteriormente. Verifica-se entre cada par pai-filho qual deles é maior, substituindo-se mutuamente caso o elemento-filho for maior que o elemento-pai, e, quando o max-heap está finalmente formado, volta-se a substituir os nós nas posições extremas (primeira e última posição) e retira-se o nó no maior index da lista.

Desta forma, os maiores elementos da lista vão ser colocados sequencialmente no final da lista - o maior na última posição, o segundo maior na penúltima, etc… - até termos um heap com apenas um elemento (n == 1), altura em que já temos o array todo ordenado como pretendemos.

A nível de complexidade computacional, o algoritmo heap sort tem complexidade O(nlog(n)) para todos os casos: pior caso, melhor caso e caso médio.

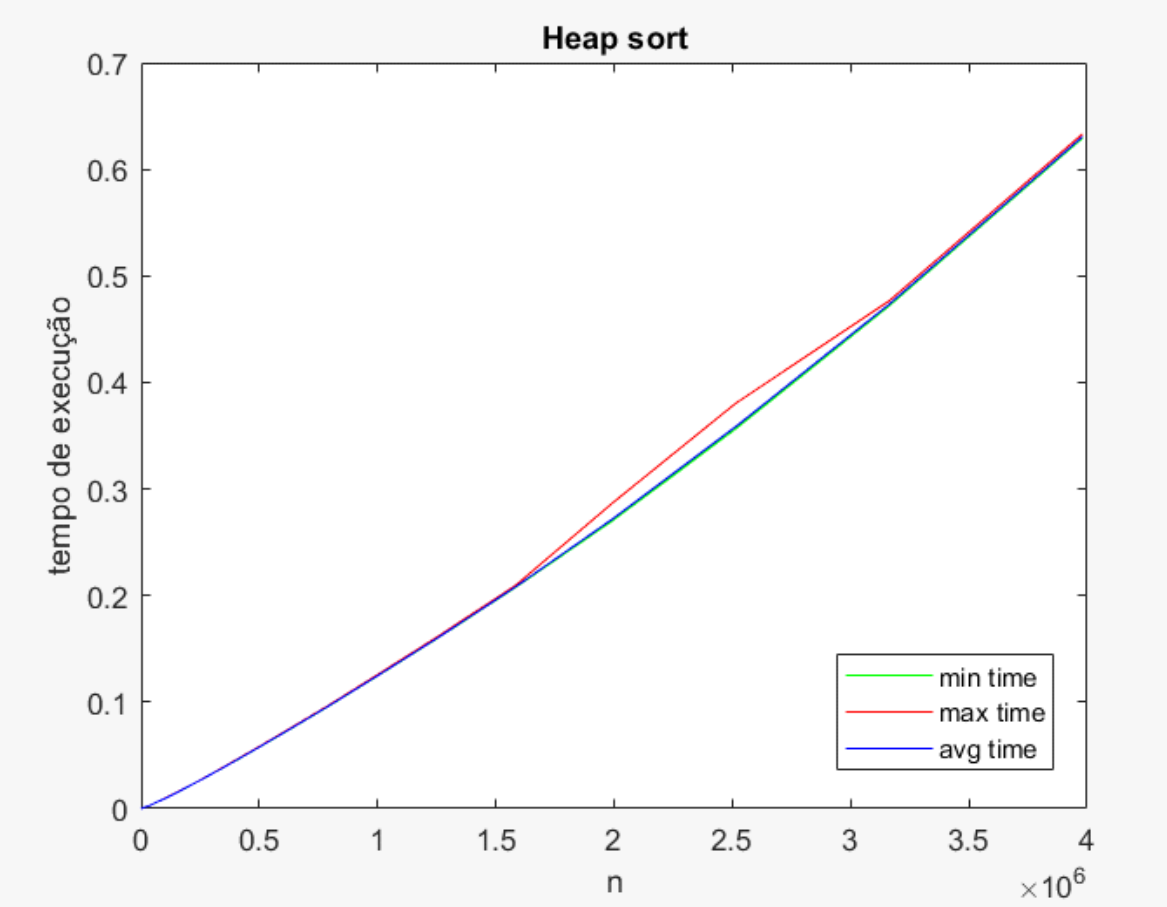
Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	5.360e-07	6.220e-07	5.682e-07	1.720e-08
13	6.050e-07	6.930e-07	6.469e-07	2.039e-08
16	6.840e-07	7.810e-07	7.302e-07	2.239e-08
20	7.960e-07	9.050e-07	8.494e-07	2.434e-08
25	9.450e-07	1.071e-06	1.006e-06	2.947e-08
32	1.182e-06	1.420e-06	1.254e-06	3.972e-08
40	1.454e-06	1.916e-06	1.576e-06	9.991e-08
50	1.819e-06	2.017e-06	1.917e-06	4.652e-08
63	2.314e-06	2.553e-06	2.432e-06	5.647e-08

79	2.973e-06	3.235e-06	3.104e-06	6.206e-08
100	3.889e-06	4.334e-06	4.039e-06	8.346e-08
126	5.051e-06	5.413e-06	5.233e-06	8.742e-08
158	6.575e-06	6.994e-06	6.783e-06	9.690e-08
200	8.681e-06	9.202e-06	8.924e-06	1.193e-07
251	1.132e-05	1.192e-05	1.160e-05	1.337e-07
316	1.467e-05	1.531e-05	1.498e-05	1.481e-07
398	1.915e-05	1.993e-05	1.953e-05	1.809e-07
501	2.512e-05	2.592e-05	2.552e-05	1.916e-07
631	3.264e-05	3.355e-05	3.310e-05	2.213e-07
794	4.249e-05	4.362e-05	4.300e-05	2.631e-07
1000	5.536e-05	5.671e-05	5.601e-05	3.154e-07
1259	7.179e-05	7.462e-05	7.269e-05	6.256e-07
1585	9.314e-05	9.493e-05	9.395e-05	4.038e-07
1995	1.210e-04	1.231e-04	1.219e-04	4.992e-07
2512	1.564e-04	1.589e-04	1.575e-04	5.716e-07
3162	2.024e-04	2.052e-04	2.036e-04	6.313e-07
3981	2.622e-04	2.702e-04	2.638e-04	1.117e-06
5012	3.383e-04	3.425e-04	3.401e-04	9.216e-07
6310	4.371e-04	4.423e-04	4.392e-04	1.099e-06
7943	5.656e-04	5.822e-04	5.681e-04	1.848e-06
10000	7.307e-04	7.684e-04	7.341e-04	4.820e-06
12589	9.450e-04	9.844e-04	9.492e-04	5.754e-06
15849	1.223e-03	1.269e-03	1.229e-03	9.119e-06
19953	1.579e-03	1.619e-03	1.585e-03	8.232e-06
25119	2.039e-03	2.085e-03	2.046e-03	1.045e-05
31623	2.636e-03	2.681e-03	2.643e-03	7.063e-06
39811	3.400e-03	3.451e-03	3.409e-03	9.757e-06
50119	4.387e-03	4.465e-03	4.401e-03	1.747e-05
63096	5.672e-03	5.732e-03	5.688e-03	1.513e-05
79433	7.327e-03	7.407e-03	7.344e-03	1.699e-05
100000	9.490e-03	9.581e-03	9.514e-03	2.015e-05
125893	1.232e-02	1.242e-02	1.234e-02	2.017e-05
158489	1.596e-02	1.610e-02	1.599e-02	2.398e-05
199526	2.067e-02	2.085e-02	2.071e-02	3.444e-05
251189	2.681e-02	2.698e-02	2.685e-02	2.721e-05
316228	3.466e-02	3.496e-02	3.473e-02	5.398e-05
398107	4.478e-02	4.532e-02	4.489e-02	9.385e-05
501187	5.791e-02	5.848e-02	5.805e-02	9.518e-05
630957	7.469e-02	7.552e-02	7.489e-02	1.374e-04
794328	9.633e-02	9.717e-02	9.654e-02	1.543e-04
1000000	1.246e-01	1.258e-01	1.248e-01	2.262e-04
1258925	1.607e-01	1.622e-01	1.610e-01	3.174e-04
1584893	2.079e-01	2.098e-01	2.084e-01	4.455e-04
1995262	2.705e-01	2.872e-01	2.722e-01	2.176e-03
2511886	3.560e-01	3.800e-01	3.581e-01	3.486e-03

3162278	4.717e-01	4.764e-01	4.732e-01	1.022e-03
3981072	6.294e-01	6.333e-01	6.307e-01	9.001e-04

Gráfico dos valores obtidos:



3.8. Rank sort

O rank sort é um algoritmo de ordenação onde cada elemento de um array é comparado com todos os outros elementos para ver qual é maior. O rank de um elemento é definido como o número total de elementos menor que esse mesmo elemento. O array é ordenado de acordo com o rank que está associado a um elemento.

Exemplo de como funciona o rank sort:

Array inicial é : [5, 4 , 12 , 8, 3, 17].

Associar a cada elemento o seu respetivo rank:

Elemento	Rank
5	3
4	2
12	5
8	4
3	1
17	6

Assim, basta ordenar por rank:

Elemento	Rank
3	1
4	2
5	3
8	4
12	5
17	6

Array final é : [3, 4 , 5 , 8, 12, 17].

Em relação à sua complexidade computacional, o pior, o médio e o melhor caso correspondem a O(n^2), devido a ter um for dentro de outro.

Comparando com outros algoritmos com a mesma complexidade computacional, por exemplo, o bubble sort, shaker sort, o código do rank sort tem mais linhas que os outros.

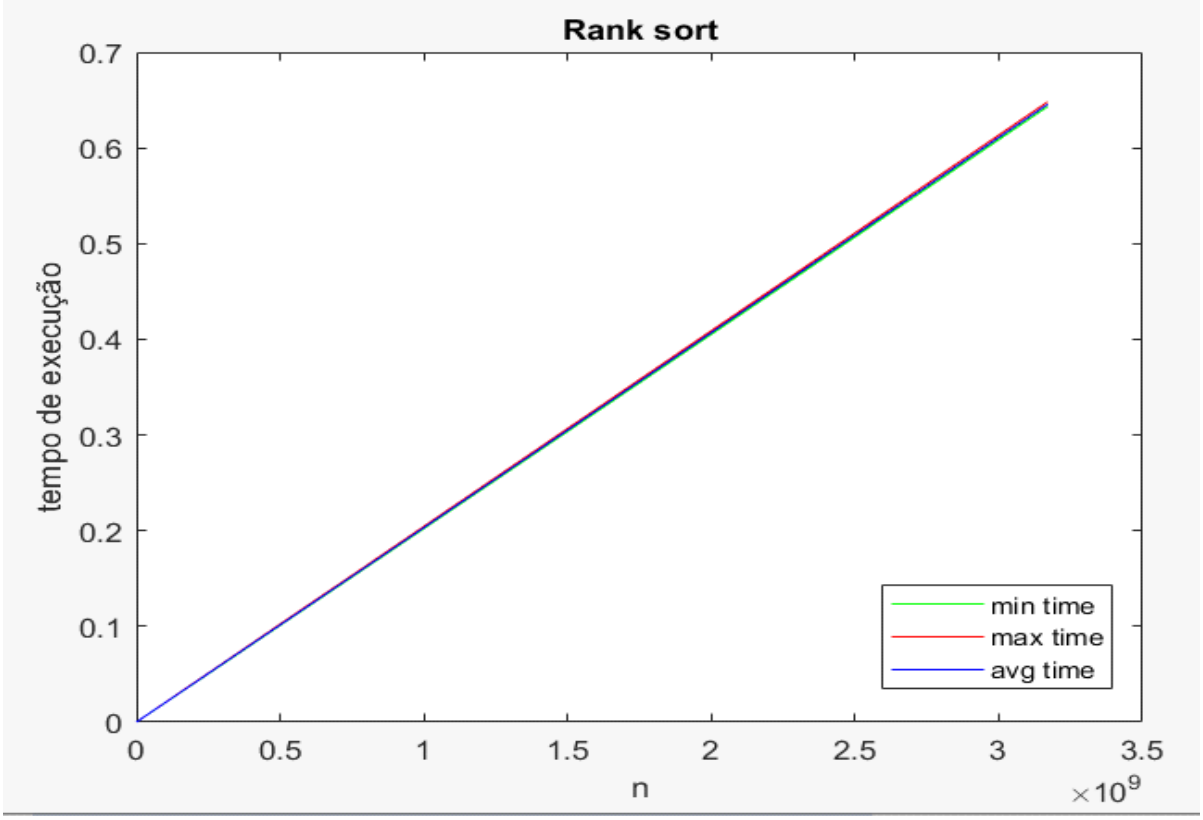
Uma desvantagem do rank sort é o facto de requerer espaço extra de memória.

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	4.960e-07	5.040e-07	4.999e-07	1.700e-09
13	5.200e-07	5.300e-07	5.245e-07	2.488e-09
16	5.510e-07	5.640e-07	5.569e-07	3.140e-09
20	6.030e-07	6.240e-07	6.121e-07	4.986e-09
25	7.110e-07	7.530e-07	7.298e-07	9.815e-09
32	8.570e-07	9.030e-07	8.778e-07	1.102e-08
40	1.089e-06	1.156e-06	1.122e-06	1.626e-08
50	1.455e-06	1.568e-06	1.504e-06	2.616e-08
63	2.040e-06	2.204e-06	2.117e-06	3.960e-08
79	2.950e-06	3.206e-06	3.071e-06	6.077e-08
100	4.443e-06	4.844e-06	4.634e-06	9.228e-08
126	6.749e-06	7.361e-06	7.056e-06	1.436e-07

158	1.034e-05	1.128e-05	1.078e-05	2.185e-07
200	1.627e-05	1.756e-05	1.690e-05	2.988e-07
251	2.533e-05	2.718e-05	2.621e-05	4.379e-07
316	3.988e-05	4.254e-05	4.119e-05	6.390e-07
398	6.313e-05	6.688e-05	6.495e-05	9.032e-07
501	9.952e-05	1.052e-04	1.022e-04	1.326e-06
631	1.578e-04	1.661e-04	1.617e-04	1.947e-06
794	2.499e-04	2.854e-04	2.558e-04	3.720e-06
1000	3.969e-04	4.169e-04	4.050e-04	4.125e-06
1259	6.298e-04	6.627e-04	6.417e-04	6.100e-06
1585	1.000e-03	1.036e-03	1.017e-03	8.121e-06
1995	1.584e-03	1.640e-03	1.610e-03	1.292e-05
2512	2.520e-03	2.609e-03	2.556e-03	1.998e-05
3162	4.005e-03	4.113e-03	4.054e-03	2.589e-05
3981	6.360e-03	6.510e-03	6.431e-03	3.494e-05
5012	1.010e-02	1.032e-02	1.020e-02	5.263e-05
6310	1.600e-02	1.632e-02	1.614e-02	7.132e-05
7943	2.539e-02	2.583e-02	2.559e-02	1.006e-04
10000	4.030e-02	4.099e-02	4.059e-02	1.459e-04
12589	6.392e-02	6.489e-02	6.436e-02	2.107e-04
15849	1.015e-01	1.029e-01	1.020e-01	2.982e-04
19953	1.609e-01	1.628e-01	1.617e-01	4.144e-04
25119	2.554e-01	2.581e-01	2.566e-01	6.416e-04
31623	4.052e-01	4.090e-01	4.070e-01	8.812e-04
39811	6.432e-01	6.482e-01	6.456e-01	1.184e-03

Gráfico dos valores obtidos:



3.9. Selection Sort

O algoritmo do Selection Sort é um algoritmo onde a ideia base acaba por ser a inversa do Bubble Sort, no Bubble Sort leva-se o valor maior para o final do array, aqui acabamos por fazer exatamente o oposto, isto é, neste algoritmo de ordenação o que se faz é colocar o menor elemento no array ao fim de cada iteração.

Contudo este método de ordenação é extremamente ineficiente pois, como ele atua é descobrir qual o índice que contém o valor mais pequeno, no final desse ciclo o valor do índice que estiver guardado na variável, caso não seja o primeiro endereço possível, trocamos para essa mesma posição o valor do menor elemento do array.

O array máximo que foi possível organizar foi, de tamanho 25119, igual ao máximo obtido pelo Bubble Sort, e verificou-se como os valores mais baixos em relação a todos os outros métodos de ordenação.

Exemplo, simples , de como funciona o Selection sort:

Array inicial é : [2, 4 ,6 , 3, 1 , 5].

1ªIteração:

[1, 4 ,6 , 3, 2 , 5].

2ªIteração:

[1, 2 ,6 , 3, 4 , 5].

3ªIteração:

[1, 2 ,3, 6, 4 , 5].

4ªIteração:

[1, 2 ,3, 4, 6 , 5].

5ªIteração:

[1, 2 ,3, 4, 5, 6].

Neste método de ordenação verificou-se uma complexidade computacional de O(n^2) para todos os casos possíveis, inclusive para o melhor caso possível.

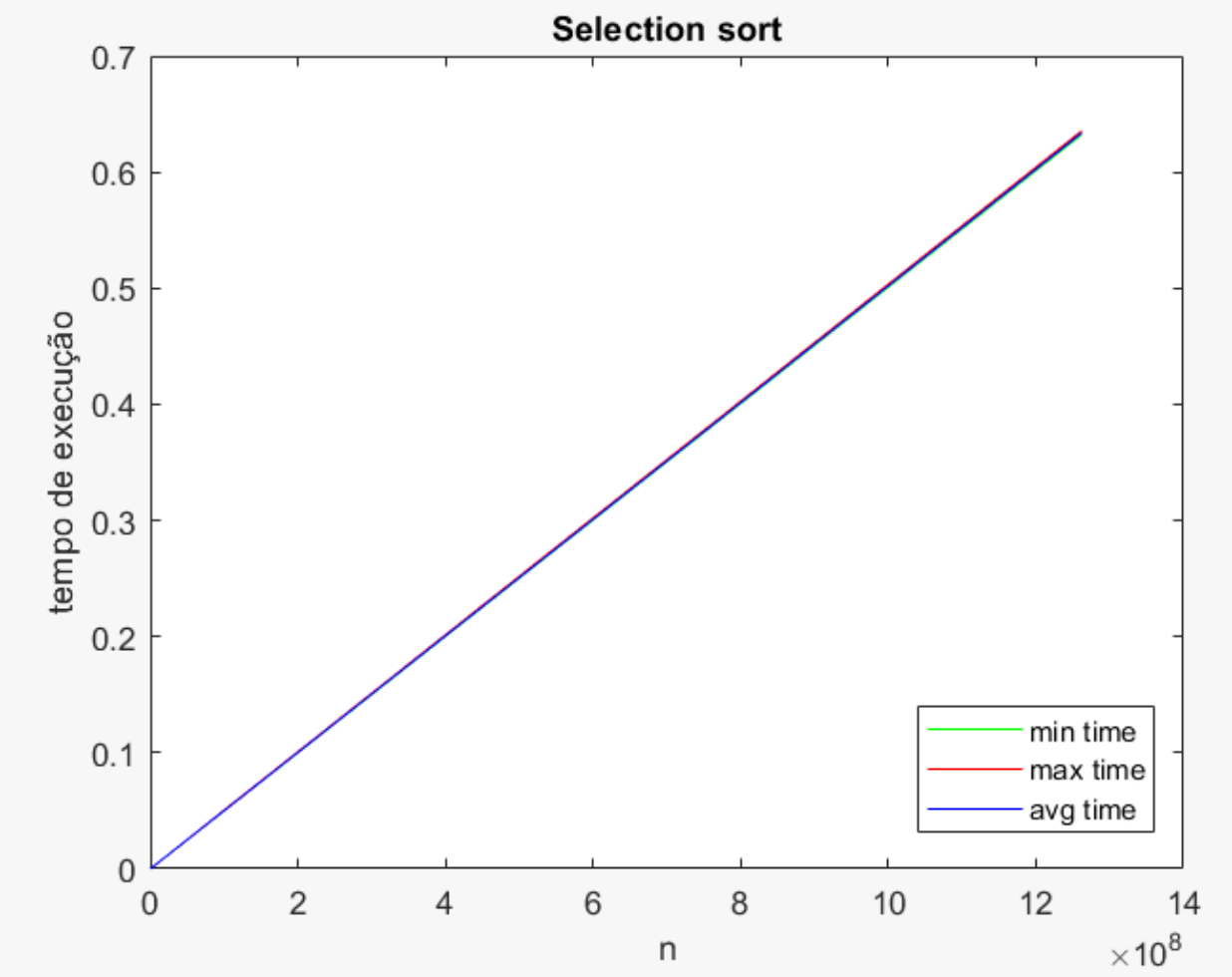
Este código ao nível da implementação acabou por verificar-se que é um código bastante compacto até visto que é possível implementar o mesmo usando cerca de 10 linhas de código C.

Tabela com os valores obtidos:

n	min time	max time	avg time	std dev
10	5.210e-07	5.530e-07	5.349e-07	7.275e-09
13	5.920e-07	6.220e-07	6.063e-07	6.893e-09
16	6.820e-07	7.120e-07	6.968e-07	7.260e-09
20	8.290e-07	8.640e-07	8.462e-07	8.470e-09
25	1.058e-06	1.095e-06	1.075e-06	8.503e-09
32	1.449e-06	1.506e-06	1.477e-06	1.082e-08
40	2.007e-06	2.090e-06	2.057e-06	1.439e-08
50	2.873e-06	2.987e-06	2.956e-06	2.281e-08
63	4.264e-06	4.471e-06	4.411e-06	4.510e-08
79	6.442e-06	6.764e-06	6.653e-06	7.657e-08
100	1.004e-05	1.053e-05	1.033e-05	1.192e-07
126	1.566e-05	1.637e-05	1.606e-05	1.717e-07
158	2.433e-05	2.538e-05	2.490e-05	2.489e-07
200	3.886e-05	4.013e-05	3.952e-05	3.081e-07
251	6.102e-05	6.281e-05	6.194e-05	4.029e-07
316	9.692e-05	9.930e-05	9.798e-05	5.092e-07
398	1.539e-04	1.573e-04	1.553e-04	6.975e-07
501	2.446e-04	2.483e-04	2.462e-04	8.323e-07
631	3.893e-04	3.936e-04	3.911e-04	1.017e-06
794	6.185e-04	6.240e-04	6.205e-04	1.168e-06
1000	9.839e-04	1.023e-03	9.873e-04	5.876e-06
1259	1.564e-03	1.603e-03	1.569e-03	8.196e-06

1585	2.485e-03	2.531e-03	2.491e-03	9.856e-06
1995	3.946e-03	3.987e-03	3.951e-03	7.876e-06
2512	6.269e-03	6.314e-03	6.277e-03	1.238e-05
3162	9.950e-03	1.002e-02	9.963e-03	1.608e-05
3981	1.580e-02	1.590e-02	1.581e-02	2.324e-05
5012	2.506e-02	2.525e-02	2.512e-02	4.420e-05
6310	3.976e-02	4.016e-02	3.984e-02	6.659e-05
7943	6.306e-02	6.367e-02	6.316e-02	1.043e-04
10000	1.000e-01	1.008e-01	1.002e-01	1.704e-04
12589	1.586e-01	1.597e-01	1.590e-01	2.720e-04
15849	2.515e-01	2.531e-01	2.520e-01	3.994e-04
19953	3.988e-01	4.008e-01	3.996e-01	4.848e-04
25119	6.323e-01	6.354e-01	6.337e-01	8.004e-04

Gráfico dos valores obtidos:



4. Comparação de algoritmos semelhantes

Antes de proceder à comparação de todos os algoritmos utilizados, achamos conveniente analisar as diferenças de funções de ordenação com complexidades semelhantes.

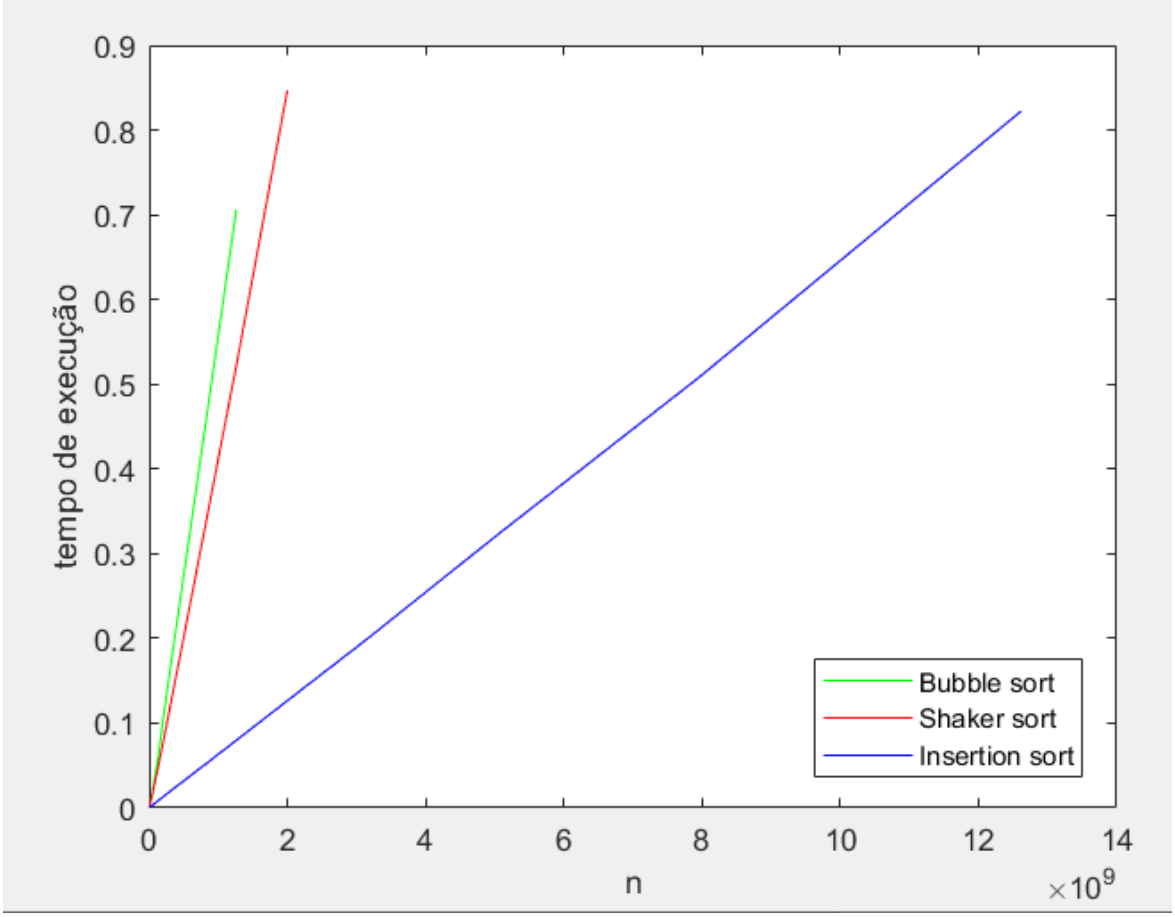
4.1. Bubble Sort vs. Shaker Sort vs. Insertion Sort

Inicialmente, vamos comparar três algoritmos com complexidades semelhantes no pior caso ($O(n)$), no melhor caso ($O(n^2)$) e em média ($O(n^2)$) que são o Bubble Sort, o Shaker Sort e o Insertion Sort.

Tendo em conta que o Shaker Sort é uma implementação melhorada do Bubble Sort, verificamos, como seria de esperar, que este primeiro é mais eficiente que o segundo. Isto acontece uma vez que além de realizar o "Up Pass", comum ao Bubble Sort, o Shaker Sort executa um passo extra: o "Down Pass". Assim, além de comparar elementos adjacentes para uma possível substituição, este vai armazenar o menor valor, dos que foram recentemente comparados, no início da lista. Apesar deste passo bónus tornar o Shaker Sort mais rápido relativamente ao Bubble Sort, ambos os algoritmos são bastante lentos em relação a outros existentes, sendo pouco usado.

Entre o Insertion Sort e ambos os algoritmos descritos anteriormente, o primeiro é mais eficiente uma vez que utiliza menos iterações até colocar os elementos na devida posição. Enquanto que o Bubble Sort coloca os máximos no final da lista depois de realizar várias comparações, o Insertion Sort permite ordenar qualquer elemento na devida posição da lista ordenada em menos do que n comparações.

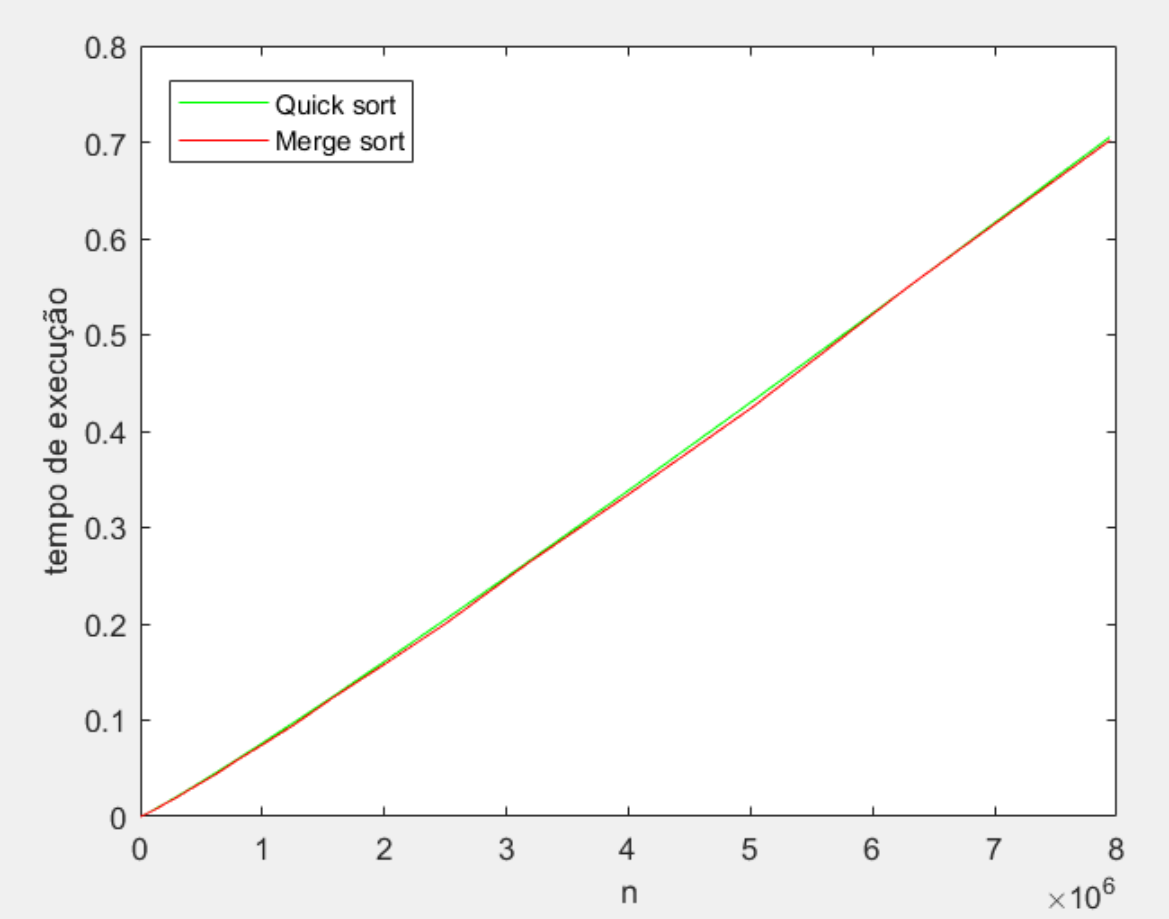
Além disso, o Insertion Sort é mais rápido uma vez que só tem uma atribuição de valor por cada comparação: guardamos na posição i o valor de i-1 caso o valor seguinte seja menor que o anterior. No caso do Bubble Sort, há três atribuições de valores por cada comparação verificada: armazenar o valor de i numa variável temporária, o index i passa a ter o valor de i-1 e, por último, o index i-1 passa a ter o valor da variável temporária. Desta forma, pode-se concluir que o Insertion Sort é o algoritmo mais rápido entre os três.



4.2. Quick Sort vs. Merge Sort

Em seguida, comparamos quick sort com o merge sort e como é possível verificar são muito idênticos em relação ao tempo de execução. O merge é melhor em algumas situações, porém não se consegue definir qual o melhor dos dois dado que são bastante idênticos.

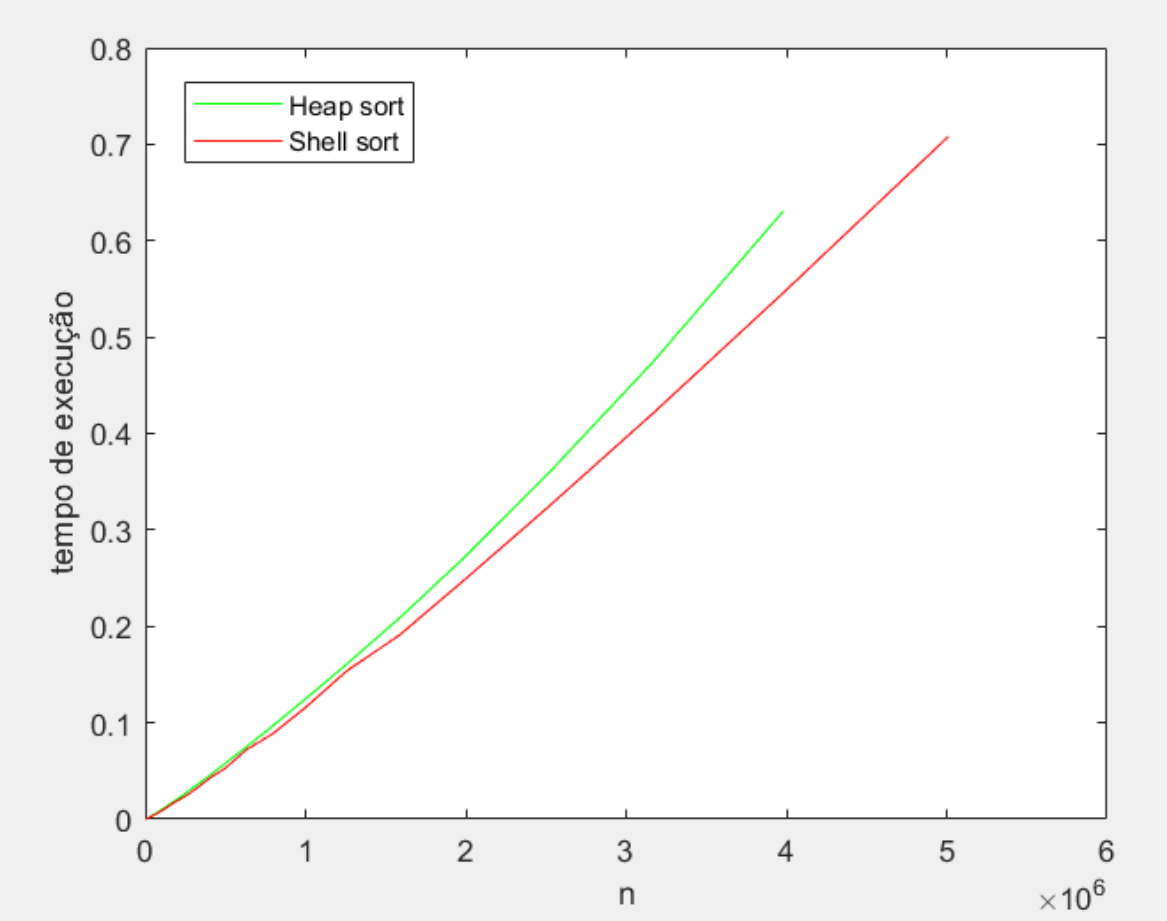
A nível de tempos de execução, usar um ou outro método é quase indiferente, iremos ter uma boa perfomance em qualquer um deles.



4.3. Heap Sort vs. Shell Sort

Como a complexidade do Shell Sort é, para nós, uma incógnita, decidimos procurar qual dos algoritmos de ordenação se assemelha mais a este e verificamos que o Heap Sort é o mais semelhante apesar de atuaram de forma bastante diferente.

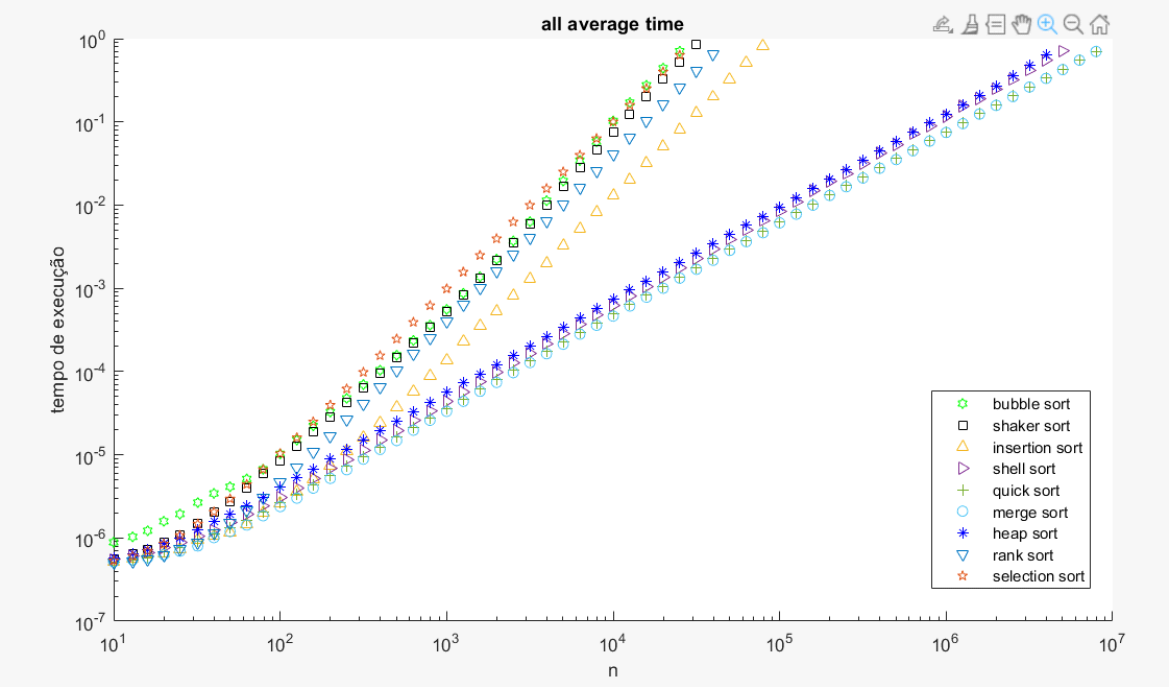
Verifica-se, através da observação do gráfico seguinte, que o shell sort acaba por ser melhor que o heap sort.


Heap sort
Shell sort
tempo de execução
n
×10^6
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
1
2
3
4
5
6

5. Comparação de todos os algoritmos

5.1. Comparação dos algoritmos

Para finalizar, vamos fazer uma análise geral dos resultados, obtidos vamos primeiro analisar os dados à grande escala, isto é, vamos analisar primeiro o gráfico onde colocamos todos os métodos de ordenação e após isto, vamos analisar, na pequena escala, ou seja, vamos analisar em detalhe para um array de tamanho 10,100, 1000 e 10000 para sabermos quais os métodos de ordenação mais rápido consoante os tamanhos do array.



Como foi possível observar pelos valores, das tabelas de cada algoritmo de ordenação, para este intervalo de tempo, alguns métodos de ordenação conseguiam chegar a valores maiores como podemos observar pelos valores das tabelas de cada algoritmo de ordenação. Para este intervalo de tempo, alguns métodos de ordenação conseguiam ordenar arrays maiores do que outros e é isso que acabamos por verificar.

Podemos verificar que a partir de valores entre o 10^4 e o 10^5, o bubble sort, o shaker sort, rank sort, insertion sort e selection sort não conseguem ordernar mais valores no intervalo de tempo de 600 segundos.

Podemos concluir que o bubble sort e o selection sort terminam no mesmo valor, ou seja o array máximo que ordenam é de tamanho 25119, no valor logarítmico a seguir termina o shaker sort com o valor de 31623, no valor seguinte que é 39811 termina o rank sort e, por último, neste intervalo termina o insertion sort com o valor de 79433.

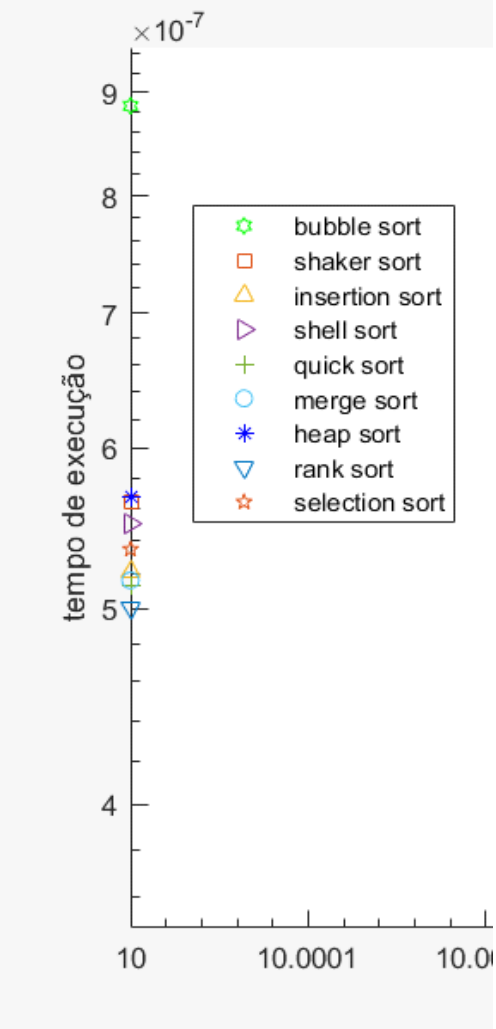
Depois, só entre o 10^6 e 10^7, terminam os outros métodos de ordenação, onde podemos verificar que o heap sort é o que termina primeiro com valor de 3981072, no valor que se segue nesta escala logarítmica, termina o shell short com valor de 5011872 e só 2 valores depois termina o quick sort e o merge sort com valor de 7943282, onde estes terminam no mesmo valor neste intervalo de tempo.

Podemos verificar que caso um método de ordenação termine mais cedo do que outro significa que esse método de ordenação é menos eficiente para ordenar arrays grandes do que aqueles que ainda conseguem ordenar arrays de tamanho maior do que onde um determinado método de ordenação terminou.

5.2. Comparação dos algoritmos para tamanhos de arrays fixos

Começando a analisar agora a eficiência de cada algoritmo de ordenação para tamanhos fixos de arrays.

Arrays com 10 elementos:



Para arrays de tamanho 10, ordenar estes arrays, à primeira vista conseguimos logo verificar que o bubble sort é muito mais ineficiente que os outros métodos de ordenação, pois este, está no eixo do y perto de valores de 9*10^-7, e os outros estão sempre em valores inferiores a 6 *10^-7.

Verificando os valores das tabelas para cada método de ordenação, neste tamanho:

Método de ordenação	Tempo de execução
Bubble Sort	8.861e-07
Shaker Sort	5.641e-07
Insertion Sort	5.209e-07
Shell Sort	5.509e-07
Quick Sort	5.131e-07
Merge Sort	5.162e-07
Heap sort	5.682e-07
Rank sort	4.999e-07
Selection sort	5.349e-07

Ordenação crescente de tempo de ordenação para arrays de tamanho 10:

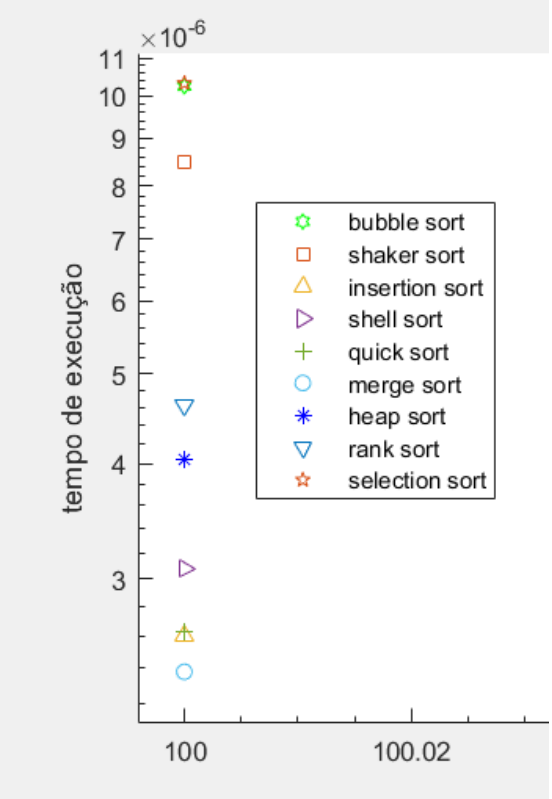
Rank , quick, merge, insertion, selection, shell, shaker, heap , bubble

Logo se pretendêssemos ordenar um array de 10 elementos, em média, o rank sort vai ser o mais eficiente, contudo a diferença não é muito significa para a maioria e de salientar que o rank sort é um método de ordenação que precisa de alocar memória extra logo nestes casos optar pelo quick ou insertion até poderia ser mais correto, o merge também precisa de alocar memória extra.

Um resultado que o grupo achou curioso é o facto de o quicksort e o merge ser mais rapido que o insertion sort,uma vez que para arrays de este tamanho o que o quicksorte o merge fazem é chamar o insertion sort, hà partida achávamos que o insertion iria ser mais rápido do que estes dois, contudo a diferença entre estes 3 é verdadeiramente pequena.

Outro apontamento a tirar é que para este tamanho do array devemos evitar sempre utilizar o bubble sort visto que este é bastante ineficiente comparando com todos os outros, mesmo estando a trabalhar numa escala de 10^-7.

Arrays com 100 elementos:



Observando a imagem relativa a este caso de estudo, podemos verificar que as diferenças começam a crescer bastante, a escala de resultados passou de 10^-7 para 10^-6 sendo que para o bubble sort e para o selection sort inclusive, já estão numa escala de 10^-5, logo podemos à partida indicar que estes dois métodos de ordenação são para ser excluídos, ou evitados a utilizar para arrays de este tamanho pois demostra-se em média mais ineficientes mas observando os valores concretos que foram obtidos pelas tabelas.

Verificando os valores das tabelas para cada método de ordenação, neste tamanho:

Métodos de ordenação	Tempos de execução
Bubble Sort	1.025e-05
Shaker Sort	8.484e-06
Insertion Sort	2.604e-06
Shell Sort	3.079e-06
Quick Sort	2.629e-06
Merge Sort	2.378e-06
Heap sort	4.039e-06
Rank sort	4.634e-06
Selection sort	1.033e-05

Ordenação crescente de tempo de ordenação para arrays de tamanho 100:

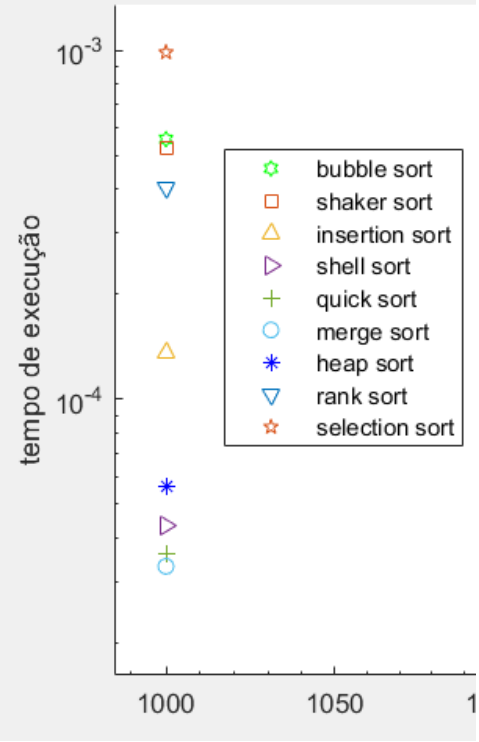
Merge, insertion, quick, shell, heap, rank, shaker, bubble e insertion.

Caso tenhamos um array de 100 elementos, e tivéssemos que escolher um método de ordenação, o melhor neste caso iriam ser entre o merge, insertion e quick pois estes são os que tem os valores de tempo mais baixos, podemos verificar isto, tanto pela imagem e também pelos valores concretos da tabela.

Mas por uma questão do código ser mais compacto neste caso optar pelo insertion sort pode acabar por ser o mais benéfico uma fez que este precisa de muito menos linhas de código do que os outros dois métodos de ordenação.

Neste caso utilizar o shaker, bubble e o selection sort iria ter um tempo de execução que já se começa a ser um bocado significativa visto que o tempo já é quase 10 vezes maior do que os 3 melhores casos logo estes três métodos, principalmente deveriam ser descartados.

Arrays com 1 000 elementos:



Observando apenas a imagem para os arrays de tamanho 1000, os 3 métodos que já eram ineficientes para o tamanho de 100 continuam a ser mas um método que anteriormente eram aceitável e inclusive para o tamanho de 10, mostrava-se como o melhor agora já começa a revelar-se muito ineficiente que é o rank sort mas observando ao pormenor os valores da tabela.

Verificando os valores das tabelas para cada método de ordenação, neste tamanho:

Método de ordenação	Tempo de execução
Bubble Sort	5.565e-04
Shaker Sort	5.267e-04
Insertion Sort	1.355e-04
Shell Sort	4.347e-05
Quick Sort	3.590e-05
Merge Sort	3.318e-05
Heap sort	5.601e-05
Rank sort	4.050e-04
Selection sort	9.873e-04

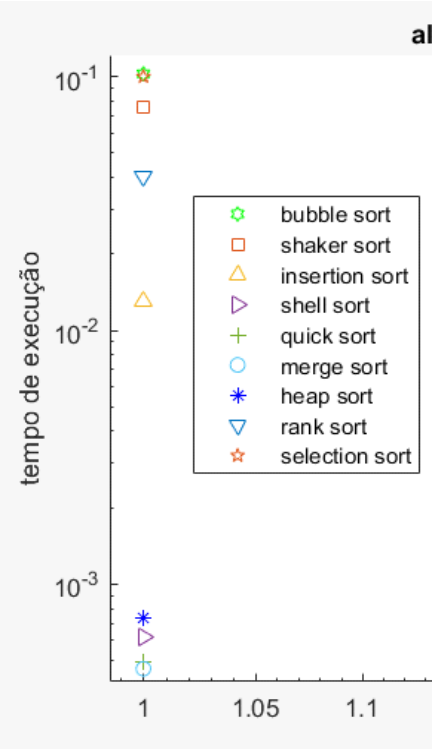
Ordenação crescente de tempo de ordenação para arrays de tamanho 1000:

Merge, quick, shell, heap, insertion, rank, shaker, bubble e selection.

Neste caso mais uma vez, os tempos de execução para o merge sort e para o quick sort estão bastante semelhantes e são estes 2 os melhores métodos de ordenação para arrays de tamanho 1000 mas caso optássemos por utilizar o shell sort ou o heap sort, ambos também iriam ser escolhas viáveis.

Escolher o insertion sort já não era uma solução tão viável mas ainda não era muito mau, mas caso escolhêssemos ordenar arrays deste tamanho com os restantes métodos de ordenação já iria ser bastante mau a nível de tempos de execução pois reletivamente aos restantes já começam a demorar um tempo considerável quando comparados com os outros.

Arrays com 10 000 elementos:



Para este tamanho do array já se começam a notar diferenças bastante grandes ao nível dos tempos de execução, onde conseguimos claramente ver pela figura que, o heap, shell, quick e merge estão com tempos inferiores a 10^-3 e os outros algoritmos já se encontram com tempos na casa do 10^-2, exemplo do insertion, rank e shaker e no caso do bubble e o selection sort inclusive já são tempos superiores a 10^-1.

Logo, sem verificar os valores das tabelas poderiamos verificar logo que pelo menos 5 métodos de ordenação há partida deviam ser logo descartados quando for necessário ordenar um array com 10000 elementos.

Verificando os valores das tabelas para cada método de ordenação, neste tamanho:

Método de ordenação	Tempo de execução
Bubble Sort	1.028e-01
Shaker Sort	7.636e-02
Insertion Sort	1.295e-02
Shell Sort	6.192e-04
Quick Sort	4.910e-04
Merge Sort	4.628e-04
Heap sort	7.341e-04
Rank sort	4.059e-02
Selection sort	1.002e-01

Ordenação crescente de tempo de ordenação para arrays de tamanho 10000:

Merge, quick, shell, heap, insertion, rank, shaker, selection e bubble.

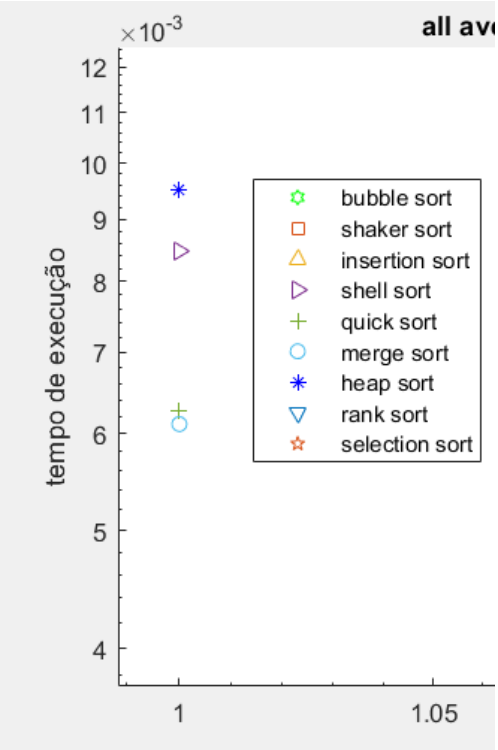
As análises que já haviam sido tiradas pelos gráficos podem agora concluir-se também pelos valores que estão registados nas tabelas, existem efetivamente 5 algoritmos de ordenação que são bastante ineficientes e que devemos descartar para ordenar arrays deste tamanho.

Mais uma vez o merge, mostra-se como o melhor algoritmo de ordenação mas como sempre até agora, com valores bastante semelhantes aqueles que são apresentados pelo quicksort.

Logo estes 2 métodos são os métodos que devem ser escolhidos para ordenar arrays deste tamanho, nunca esqueçando que agora vamos estar a trabalhar com arrays bastante grandes logo o merge sort vai começar a necessitar de alguma memória extra, que pode fazer com que seja preferível o uso do quick sort em vez de este método.

Caso o programador opte por utilizar o shell sort, ou o heap sort, é preferível que o mesmo escolha o shell sort, visto que este é ligeiramente mais rápido mas escolher o heap sort era uma escolha aceitável também.

Arrays com 100 000 elementos:



Para arrays deste tamanho, os 5 métodos de ordenação que já haviam demostrado ser menos eficientes para arrays de tamanho de 10000 para arrays de 100000, já não conseguem ordenar o array no tempo de 600 que havia sido definido quando se compilou o programa.

E para este caso começa a ser cada vez mais notória, que o algoritmo do merge sort e do quick sort são os mais rápidos mas vamos primeiro observar os resultados obtidos que estão registados nas tabelas.

Verificando os valores das tabelas para cada método de ordenação, neste tamanho:

Método de ordenação	Tempo de execução
Bubble Sort	No time
Shaker Sort	No time
Insertion Sort	No time
Shell Sort	8.466e-03
Quick Sort	6.267e-03
Merge Sort	6.111e-03
Heap sort	9.514e-03
Rank sort	No time
Selection sort	No time

Ordenação crescente de tempo de ordenação para arrays de tamanho 100000:

Merge, quick, shell, heap.

Podemos verificar que o que dava para ver pela imagem se verifica que o quick e o merge sort são os algoritmos de ordenação mais rápidos para arrays deste tamanho, com diferenças de tempo bastante pequenas, logo mais uma vez escolher um método de ordenação ou escolher o outro pode ser bastante semelhante, o resultado obtido para ordenar o array.

Escolher o Shell Sort, ou o Heap sort, as diferenças ainda não são muito significativas contudo já se consegue notar algumas diferenças mas consideramos que estas diferenças não tornam o código muito mais ineficiente caso utilizarmos este método em vez dos outros que tem um tempo de execução melhor.

6. Conclusão

Podemos verificar que regra geral os algoritmos do merge sort e do quick sort são os algoritmos que devemos utilizar, pois são quase sempre estes os que são os mais eficientes para quase todos os tamanhos dos arrays.

Contudo para arrays muito pequenos, podemos também utilizar o rank sort que demonstou ser um bom algortimo de ordenação para arrays até 10 elementos ou também para arrays pequenos até cerca de 100 elementos podemos utilizar o insertion sort que é um algortimo de ordenação bastante compacto e que atua muito bem para este tamanho de array.

A outra conclusão que conseguimos tirar é que os algoritmos do bubble sort, shaker sort e do selection sort, são 3 algoritmos de ordenação bastante inefientes e em nenhuma das situações que foram aboradadas por nos devemos utilizar estes casos, pois temos sempre mais opções que são melhores tanto a nível de tempo de execução tanto a nível do codigo ser mais compacto