universidadede
de aveiro

Algoritmos e Estruturas de Dados
Licenciatura em Engenharia Informática
Ano Letivo 2020/2021 2ºAno 1º Semestre

Relatório do Trabalho Prático 2

Estudo sobre rotinas de ordenação

Trabalho realizado por:
Diogo Cruz (98595) - 33.3%
Gonçalo Leal (98008) - 33.3%
Sophie Pousinho (97814) - 33.3%

Índice

Introdução

Neste trabalho iremos analisar diferentes rotinas de ordenação: em que consistem, exemplos, complexidades computacionais e ainda comparação de resultados considerando diferentes números de valores.

É bom referir que nas considerações da complexidade computacional em maior parte dos casos, o melhor caso é os elementos estarem já ordenados corretamente e o pior caso é os elementos estarem ordenados de forma contrária à desejada. No entanto, na grande maioria dos casos os dados estão desordenados e são estes os casos que pretendemos analisar, pois são estes que realmente permitem avaliar a prestação dos algoritmos.

Dentro dos algoritmos escolhidos há alguns mais simples de implementar que outros, há outros que são mais eficientes. Embora todos sejam eficazes, pois cumprem a sua função que é a de ordenar um conjunto de dados, o nosso objetivo é encontrar o algoritmo com a melhor relação entre eficiência e dificuldade de implementação.

Bubble Sort

Algoritmo

O Bubble Sort funciona de forma recursiva. Consiste em ordenar um array tendo em conta um ponteiro de first e um ponteiro de one after last, ou seja, o intervalo a ordenar é dos índices [first, one after last[. O algoritmo traduz-se em ordenar cada valor do array considerando um valor e o seu valor adjacente, se tiverem na ordem errada trocam de posições.
Se estivermos a considerar uma ordenação decrescente, isso significa que o menor valor irá ficar na última posição logo na primeira passagem completa. No caso de ordem crescente, na última posição fica o maior valor do array. Assim, na segunda passagem, já não teremos de considerar o último índice porque já está ordenado.

Por exemplo: Considerando o array com os valores [6,2,8,4,10] e uma ordenação crescente.

Início da 1ª passagem

1. [**6**,**2**,8,4,10] (6>2, trocam-se)
2. [2,**6**,**8**,4,10] (6<8, mantêm-se)
3. [2,6,**8**,**4**,10] (8>4, trocam-se)
4. [2,6,4,**8**,**10**] (8<10, mantêm-se)

1ª passagem completa
Início da 2ª passagem

5. [**2**,**6**,4,8,10] (2<6, mantêm-se)
6. [2,**6**,**4**,8,10] (6>4, trocam-se)
7. [2,4,**6**,**8**,10] (6<8, mantêm-se)
8. [2,4,6,**8**,**10**] (8<10, mantêm-se)

2ª passagem completa e array ordenado de modo crescente

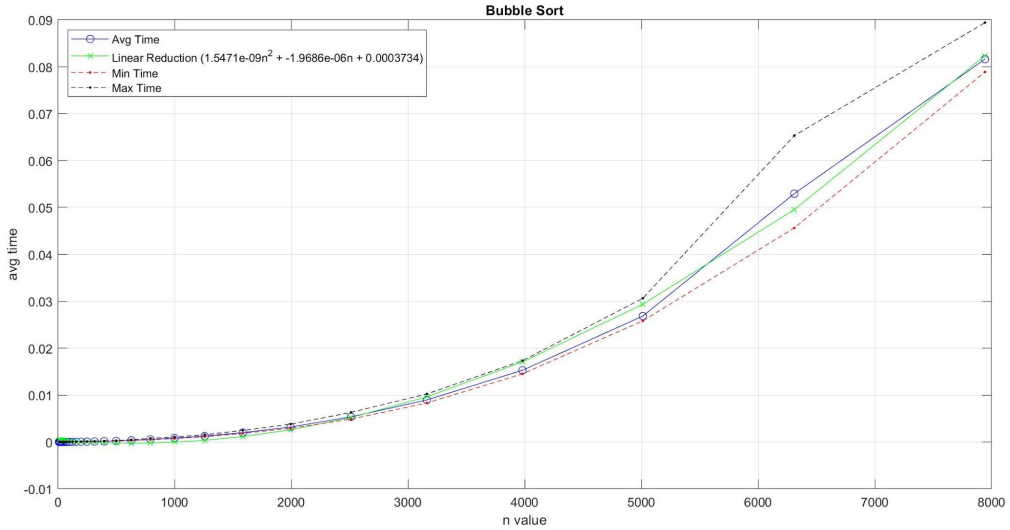
Complexidades computacionais

No melhor caso: $O(n)$
No caso médio: $O(n^2)$
No pior caso: $O(n^2)$

n	min time	max time	avg time	std dev
10	7.270e-07	8.720e-07	7.861e-07	3.194e-08
13	8.390e-07	1.155e-06	9.356e-07	6.120e-08
16	9.840e-07	1.214e-06	1.095e-06	5.396e-08
20	1.226e-06	1.526e-06	1.372e-06	7.067e-08
25	1.583e-06	2.129e-06	1.784e-06	1.128e-07
32	1.952e-06	2.904e-06	2.316e-06	2.182e-07
40	2.567e-06	3.205e-06	2.864e-06	1.440e-07
50	3.682e-06	5.045e-06	4.134e-06	2.626e-07
63	5.389e-06	7.243e-06	5.985e-06	3.339e-07
79	7.832e-06	1.403e-05	8.733e-06	6.502e-07
100	1.221e-05	2.028e-05	1.377e-05	1.374e-06
126	1.828e-05	3.681e-05	2.117e-05	2.524e-06
158	2.668e-05	4.683e-05	2.984e-05	3.524e-06
200	4.005e-05	6.676e-05	4.473e-05	5.489e-06
251	5.940e-05	9.956e-05	6.652e-05	8.332e-06
316	8.868e-05	1.408e-04	9.762e-05	1.114e-05
398	1.306e-04	2.121e-04	1.457e-04	1.727e-05
501	1.950e-04	2.989e-04	2.202e-04	2.533e-05
631	2.988e-04	4.791e-04	3.373e-04	3.755e-05
794	4.644e-04	7.729e-04	5.361e-04	6.694e-05
1000	7.148e-04	1.057e-03	8.137e-04	7.532e-05
1259	1.114e-03	1.527e-03	1.230e-03	9.483e-05
1585	1.812e-03	2.455e-03	1.985e-03	1.479e-04
1995	2.920e-03	3.785e-03	3.188e-03	1.976e-04
2512	4.869e-03	6.302e-03	5.347e-03	3.419e-04
3162	8.307e-03	1.024e-02	8.957e-03	4.153e-04
3981	1.452e-02	1.733e-02	1.528e-02	5.594e-04
5012	2.582e-02	3.063e-02	2.684e-02	7.559e-04
6310	4.564e-02	6.532e-02	5.295e-02	5.461e-03
7943	7.889e-02	8.942e-02	8.162e-02	1.962e-03

Tab. 1 Tabela de valores para Bubble Sort

Gráfico



Img. 1 Bubble Sort

O gráfico da imagem 1 apresenta as curvas dos casos mínimo, médio e máximo do bubble sort e a curva da linearização do caso médio (pelo método dos mínimos quadrados). Tal como, também, se pode observar na tabela, não há resultados para valores de n superiores a 8000. Isto deve-se à condição de paragem do programa que para a execução do método de ordenação quando o tempo médio ultrapassa os 60 segundos.

A expressão obtida com a linearização foi de ordem n^2 que é o esperado para o caso médio. No entanto, podemos observar que para o caso mínimo a curva é semelhante à da linearização, ou seja, o caso mínimo será igualmente de ordem n^2. Isto verifica-se porque o caso mínimo teórico acontece quando o array já se encontra ordenado. Como o algoritmo de medição dos tempos de execução cria arrays aleatórios então dificilmente irá acontecer que o tempo mínimo corresponda ao caso mínimo teórico, porque o array não estará ordenado por defeito.

Heapsort

Algoritmo

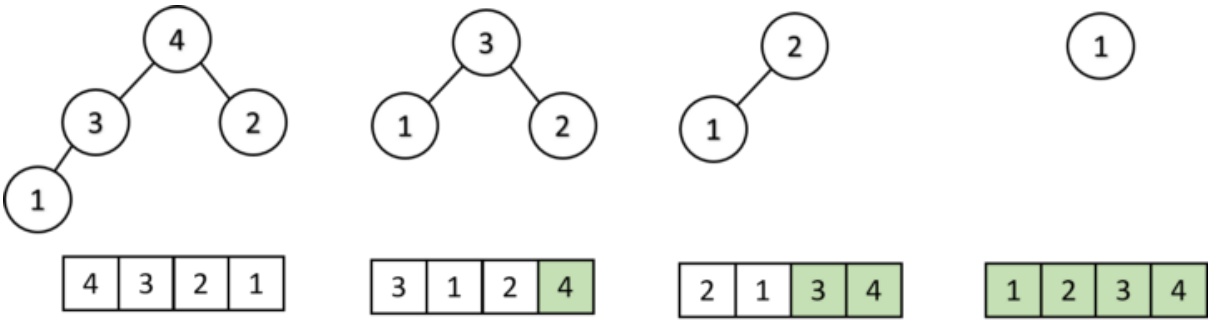
A Heapsort é uma rotina de ordenação baseada na Binary Heap. A Binary Heap é um árvore binária completa, isto é, é uma árvore com todos os níveis em profundidade ocupados com possível exceção do último; e, por outro lado, possui uma ordenação do tipo Max-Heap (quando o nó-pai é maior que os nós-filho) ou Min-Heap (quando o nó-pai é menor que os nós-filho).
Vamos considerar uma rotina de ordenação crescente. Para ordenar iremos considerar 2 passos:

1. Construir o max-heap com os valores que temos no array;
2. Ir “retirando” sempre o maior número, que está na root do heap. Ao retirarmos, ficamos com uma posição livre no final do heap e colocamos lá esse valor.

Para uma ordenação decrescente, temos de:

1. Construir o min-heap com os valores do array;
2. Retirar o valor menor, que irá estar na root do heap, e colocá-lo na última posição que irá estar livre.

Por exemplo: Consideremos o exemplo da figura e uma ordenação crescente.



Neste caso, o array já se encontra na forma de Max-Heap, os nós-filhos são menores que o nó-pai, 4 é maior que 3 e 2 e 3 é maior que 1. De seguida, colocamos o maior número no final do array (4). “Esquecemo-nos” do 4 e criamos a nova Max-Heap em que 3 passa a ser a root e 1 e 2 são seus filhos. O valor 3 é o maior, logo, passa para o fim do array e é “esquecido” e passa-se a considerar 1 e 2. Aplica-se o mesmo raciocínio.

Complexidades computacionais

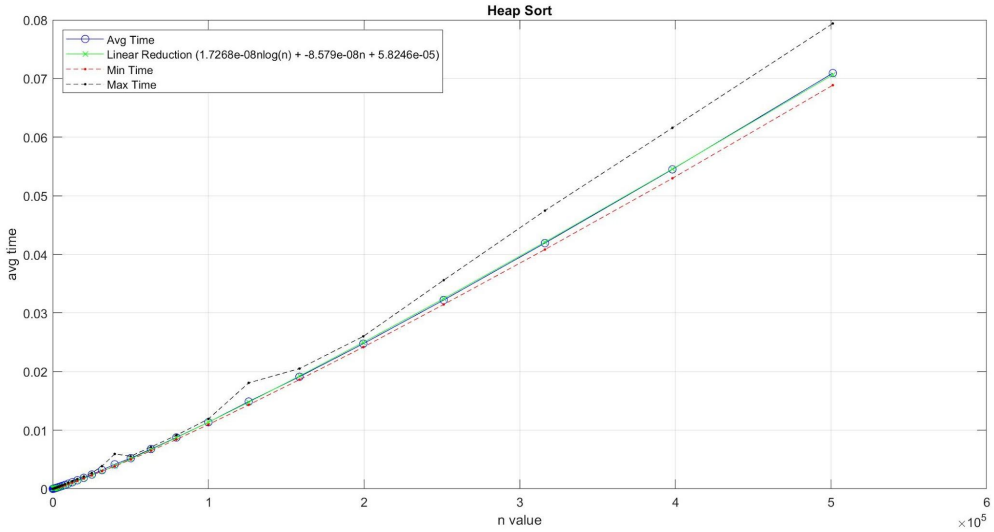
No melhor caso: *O(n log n)*
No caso médio: *O(n log n)*
No pior caso: *O(n log n)*

n	min time	max time	avg time	std dev
10	6.160e-07	7.110e-07	6.544e-07	2.112e-08
13	6.870e-07	7.900e-07	7.366e-07	2.509e-08
16	7.800e-07	1.092e-06	8.589e-07	6.312e-08
20	9.090e-07	1.185e-06	9.810e-07	4.431e-08
25	1.089e-06	1.264e-06	1.167e-06	4.145e-08
32	1.338e-06	1.539e-06	1.425e-06	4.633e-08
40	1.637e-06	1.867e-06	1.746e-06	5.229e-08
50	2.051e-06	2.613e-06	2.185e-06	7.950e-08
63	2.619e-06	2.957e-06	2.774e-06	7.784e-08
79	3.339e-06	4.305e-06	3.530e-06	1.264e-07
100	4.341e-06	4.817e-06	4.547e-06	1.064e-07
126	5.628e-06	6.301e-06	5.887e-06	1.392e-07
158	7.329e-06	9.966e-06	7.646e-06	2.938e-07
200	9.632e-06	1.265e-05	1.001e-05	3.994e-07
251	1.257e-05	2.594e-05	1.308e-05	8.665e-07
316	1.638e-05	3.094e-05	1.713e-05	1.697e-06
398	2.129e-05	3.573e-05	2.220e-05	2.070e-06
501	2.778e-05	4.645e-05	2.920e-05	3.218e-06
631	3.614e-05	5.741e-05	3.795e-05	3.777e-06
794	4.692e-05	7.742e-05	4.996e-05	5.903e-06
1000	6.112e-05	9.814e-05	6.488e-05	7.262e-06
1259	7.920e-05	1.256e-04	8.466e-05	9.238e-06
1585	1.026e-04	1.540e-04	1.093e-04	1.161e-05
1995	1.332e-04	1.959e-04	1.432e-04	1.515e-05
2512	1.725e-04	2.400e-04	1.864e-04	1.724e-05
3162	2.232e-04	3.138e-04	2.438e-04	2.209e-05
3981	2.889e-04	3.824e-04	3.134e-04	2.236e-05
5012	3.730e-04	4.968e-04	4.041e-04	3.002e-05
6310	4.819e-04	6.192e-04	5.183e-04	3.287e-05
7943	6.241e-04	8.293e-04	6.773e-04	3.884e-05
10000	8.140e-04	1.061e-03	8.784e-04	5.173e-05
12589	1.054e-03	1.322e-03	1.129e-03	5.556e-05
15849	1.366e-03	1.676e-03	1.461e-03	6.675e-05
19953	1.768e-03	2.105e-03	1.879e-03	7.403e-05
25119	2.304e-03	2.712e-03	2.442e-03	9.039e-05

31623	3.009e-03	3.868e-03	3.207e-03	1.532e-04
39811	3.855e-03	5.952e-03	4.177e-03	3.544e-04
50119	4.994e-03	5.563e-03	5.236e-03	1.357e-04
63096	6.499e-03	7.135e-03	6.789e-03	1.493e-04
79433	8.423e-03	9.130e-03	8.762e-03	1.607e-04
100000	1.097e-02	1.192e-02	1.136e-02	1.873e-04
125893	1.433e-02	1.808e-02	1.489e-02	6.238e-04
158489	1.863e-02	2.051e-02	1.912e-02	3.068e-04
199526	2.418e-02	2.601e-02	2.476e-02	3.454e-04
251189	3.145e-02	3.560e-02	3.221e-02	5.657e-04
316228	4.083e-02	4.746e-02	4.192e-02	9.725e-04
398107	5.297e-02	6.158e-02	5.450e-02	1.425e-03
501187	6.886e-02	7.939e-02	7.092e-02	1.876e-03

Tab. 2 Tabela de valores para Heapsort

Gráfico



Img. 2 Heap Sort

Na imagem 2 podemos observar o gráfico que apresenta as curvas dos casos mínimo, médio e máximo do Heap Sort e a curva da linearização do caso médio (pelo método dos mínimos quadrados). Com este método já é possível ordenar um array com mais de 400 mil elementos em menos de 60 segundos em média. Sendo que o último valor de n testado (501187) já ultrapassa os 60 segundos.

Este método tem complexidade computacional para os melhor, pior e caso médio na ordem de *n log n.* Pela análise do gráfico é possível constatar isso, porque a expressão da redução linear é da mesma

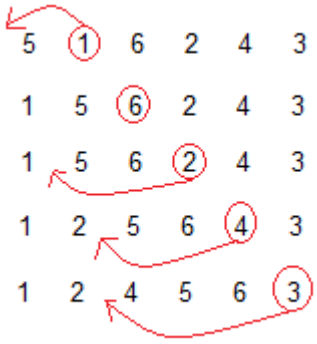
ordem. E, embora tenham taxas de crescimento diferentes devido aos coeficientes que multiplicam as incógnitas serem diferentes, tanto o gráfico do tempo mínimo como o do tempo máximo acompanham a curva da redução linear.

Insertion Sort

Algoritmo

Nesta rotina de ordenação vamos dividir o array na parte ordenada e na parte desordenada. Inicialmente a parte ordenada é apenas o primeiro elemento do array. Depois, vamos buscar cada um dos outros elementos do array um a um e colocá-los na parte ordenada na suas posições corretas. Funciona como se tivéssemos a considerar dois arrays diferentes, um com os valores desordenados e outro onde os queremos colocar no sítio certo. Por cada valor que queremos inserir no array ordenado, temos de fazer uma passagem por cada valor da parte ordenada para comparar valores e determinar a posição do novo elemento.

Por exemplo: Queremos fazer uma ordenação crescente dos elementos 5 1 6 2 4 3



Inicialmente vamos considerar como se o array só tivesse o 5 e queríamos introduzir o 1. Como 5>1, passamos a ter [1 5]. Depois queremos adicionar o 6, como 6>1 e 6>5, temos [1 5 6]. Depois adicionamos o 2, como 1<2, 5>2 e 6>2, o array fica [1 2 5 6]. O mesmo processo vai ocorrer para a inserção de 4 e de 3.

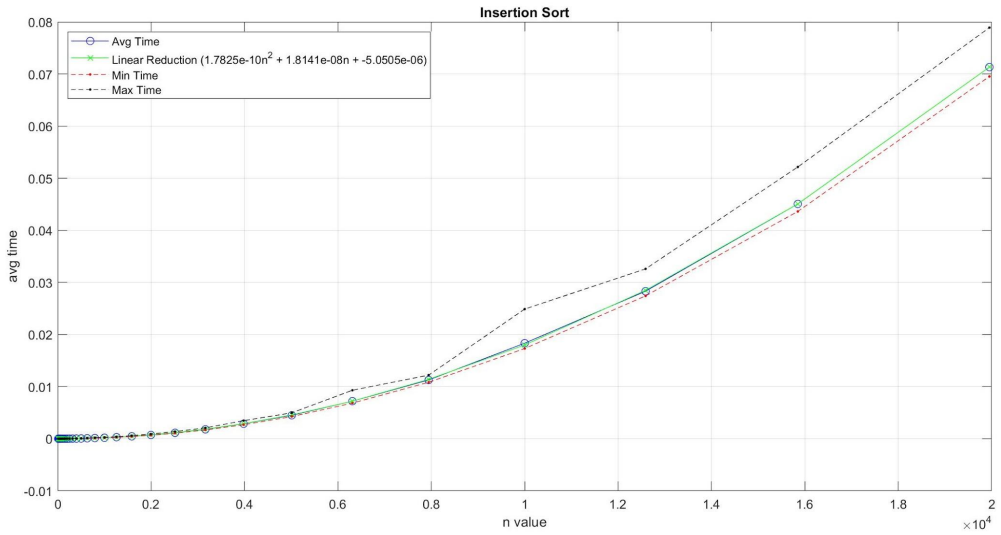
Complexidades computacionais

No melhor caso: $O(n)$
No caso médio: $O(n^2)$
No pior caso: $O(n^2)$

n	min time	max time	avg time	std dev
10	5.770e-07	7.720e-07	6.199e-07	4.259e-08
13	6.140e-07	6.780e-07	6.418e-07	1.493e-08
16	6.520e-07	7.250e-07	6.840e-07	1.684e-08
20	7.120e-07	8.150e-07	7.491e-07	2.085e-08
25	7.910e-07	8.990e-07	8.350e-07	2.442e-08
32	9.160e-07	1.060e-06	9.827e-07	3.141e-08
40	1.089e-06	1.448e-06	1.177e-06	6.683e-08
50	1.322e-06	1.542e-06	1.422e-06	4.811e-08
63	1.665e-06	1.947e-06	1.776e-06	5.618e-08
79	2.257e-06	3.551e-06	2.468e-06	2.044e-07
100	3.548e-06	5.219e-06	4.644e-06	3.511e-07
126	4.129e-06	5.305e-06	4.464e-06	1.846e-07
158	5.985e-06	1.004e-05	6.575e-06	5.627e-07
200	8.710e-06	1.165e-05	9.365e-06	3.788e-07
251	1.284e-05	2.431e-05	1.402e-05	1.491e-06
316	1.931e-05	3.671e-05	2.137e-05	3.050e-06
398	2.938e-05	4.945e-05	3.185e-05	3.300e-06
501	4.505e-05	7.424e-05	4.925e-05	5.157e-06
631	7.069e-05	1.349e-04	8.266e-05	1.670e-05
794	1.086e-04	2.038e-04	1.267e-04	2.375e-05
1000	1.695e-04	2.646e-04	1.895e-04	2.046e-05
1259	2.666e-04	3.759e-04	2.960e-04	2.375e-05
1585	4.191e-04	5.882e-04	4.594e-04	3.076e-05
1995	6.626e-04	9.057e-04	7.240e-04	4.657e-05
2512	1.060e-03	1.376e-03	1.144e-03	6.288e-05
3162	1.681e-03	2.082e-03	1.803e-03	8.526e-05
3981	2.687e-03	3.475e-03	2.874e-03	1.409e-04
5012	4.261e-03	4.972e-03	4.503e-03	1.480e-04
6310	6.833e-03	9.302e-03	7.208e-03	3.339e-04
7943	1.084e-02	1.221e-02	1.128e-02	2.353e-04
10000	1.731e-02	2.488e-02	1.834e-02	1.525e-03
12589	2.741e-02	3.260e-02	2.831e-02	7.598e-04
15849	4.362e-02	5.215e-02	4.506e-02	1.620e-03
19953	6.953e-02	7.890e-02	7.132e-02	1.757e-03

Tab. 3 Tabela de valores para Insertion Sort

Gráfico



Img. 3 Insertion Sort

Na imagem 3 podemos observar o gráfico que apresenta as curvas dos casos mínimo, médio e máximo do Insertion Sort e a curva da linearização do caso médio (pelo método dos mínimos quadrados). Com este método é possível ordenar um array com 18 mil elementos em menos de 60 segundos em média. Sendo que o último valor de n testado (19953) já ultrapassa os 60 segundos em média.
Este método tem complexidade computacional *n* para o melhor caso. No entanto, pela observação do gráfico, vemos que a curva do tempo mínimo acompanha a curva do tempo médio e da regressão linear, o que quer dizer que tem uma expressão de ordem n^2 e não de ordem *n.* Isto acontece porque o melhor caso acontece quando o array já se encontra ordenado e o algoritmo que testa os métodos de ordenação cria os arrays que irão ser ordenados de forma aleatória e muito dificilmente este estaria ordenado por defeito.

Mergesort

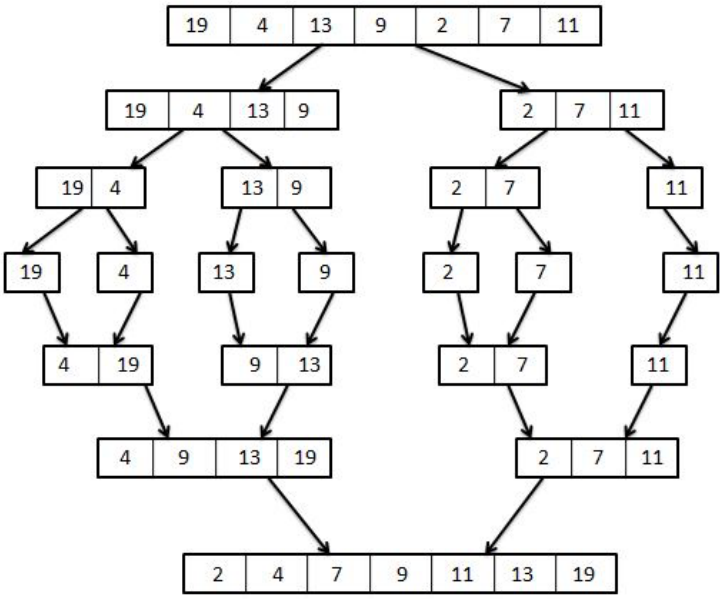
Algoritmo

Tal como o Quicksort, esta rotina de ordenação é baseada no Divide and Conquer.

O Divide and Conquer é um algoritmo que pretende resolver um problema de forma recursiva usando uma estratégia genérica: **(Divide**) Se o problema for suficientemente pequeno aplicamos o algoritmo diretamente. Se o problema for demasiado grande, subdividimos o problema em problemas mais pequenos do mesmo tipo (subdivisão, normalmente, inteira, e partes de tamanho semelhante). **(Conquer**) Para resolver os problemas mais pequenos usa-se o mesmo algoritmo recursivamente. (**Combine**) A fase final é combinar os resultados, constrói-se a solução para o problema combinando as soluções para problemas mais pequenos.

Este algoritmo funciona de forma semelhante ao Quicksort (descrito a seguir) no sentido em que ordena o array, dividindo-o em 3 partes consoante um pivot, os valores menores que o pivot ficam atrás dele, os iguais junto dele e os maiores depois dele. No entanto, no Mergesort, o *pivot* não se escolhe, é o valor do meio do array. Por isso, calcula-se o index do meio e ordenam-se as duas partes do array recursivamente, isto é, escolhendo para cada uma das partes do array um novo *pivot* para cada novo “sub-array” e fazendo as mesmas operações sucessivamente. Depois juntam-se as duas partes ordenadas para ter a ordenação do conjunto todo. Faz-se uma passagem final de *O(1)* mas é necessário usar uma zona de memória auxiliar porque, no pior caso, os números do lado esquerdo, são maiores do que o da direita.

Uma desvantagem deste algoritmo é, portanto, a utilização de mais espaço de memória.



Por exemplo:

Queremos ordenar de forma crescente. Começamos por dividir o array em duas partes, como temos 7 elementos, um lado fica com 4 (19 4 13 9) e o outro com 3 (2 7 11). Depois voltamos a dividir as metades. A metade com 4 elementos passa a dois conjuntos de 2 elementos (4 19 e 9 13) e a metade com 3 elementos passa a dois conjuntos, um com 2 elementos (2 7) e outro com um (11). Voltamos a dividir as novas metades e passamos a ter “sub-arrays” com 1 elemento. Depois voltam-se a juntar os valores considerando o pivot (elemento na posição central de cada array), os valores menores ficam atrás e os maiores à frente, isto é feito sucessivamente até o array estar completo.

Complexidades computacionais

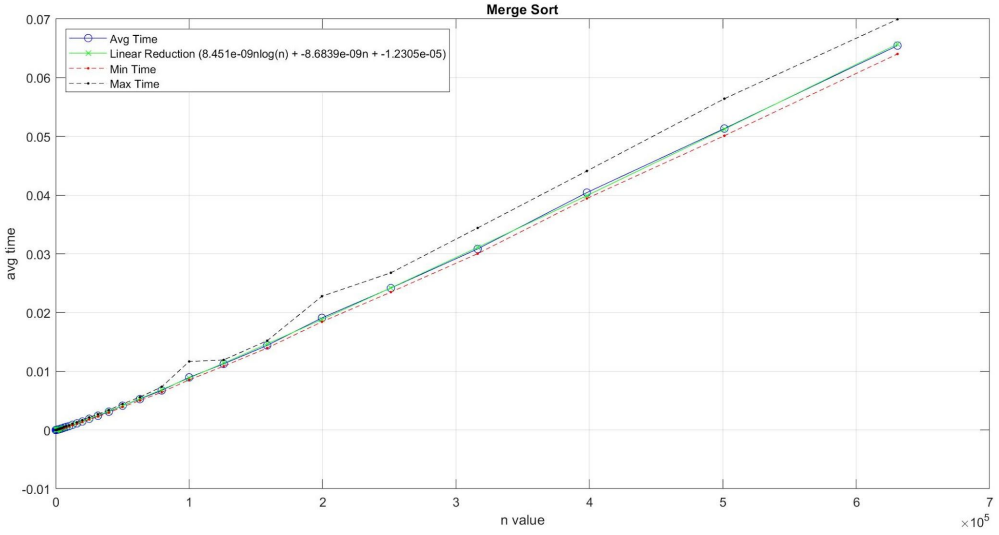
No melhor caso: *O(n log n)*
No caso médio: *O(n log n)*
No pior caso: *O(n log n)*

n	min time	max time	avg time	std dev
10	5.740e-07	6.320e-07	5.977e-07	1.333e-08
13	6.160e-07	6.690e-07	6.405e-07	1.258e-08
16	6.540e-07	7.200e-07	6.830e-07	1.565e-08
20	7.230e-07	9.770e-07	7.843e-07	5.706e-08
25	7.950e-07	9.010e-07	8.411e-07	2.374e-08
32	9.270e-07	1.055e-06	9.863e-07	2.909e-08
40	1.196e-06	1.342e-06	1.266e-06	3.433e-08
50	1.422e-06	2.056e-06	1.555e-06	1.523e-07
63	1.738e-06	2.244e-06	1.848e-06	8.074e-08
79	2.277e-06	3.247e-06	2.429e-06	1.534e-07
100	2.961e-06	3.317e-06	3.104e-06	7.554e-08
126	3.731e-06	4.178e-06	3.914e-06	9.764e-08
158	4.991e-06	5.875e-06	5.224e-06	1.461e-07
200	6.619e-06	8.340e-06	6.867e-06	1.939e-07
251	8.440e-06	1.192e-05	8.810e-06	4.301e-07
316	1.180e-05	1.494e-05	1.218e-05	3.283e-07
398	1.507e-05	2.614e-05	1.570e-05	9.898e-07
501	1.930e-05	3.586e-05	2.082e-05	2.778e-06
631	2.602e-05	4.184e-05	2.741e-05	1.969e-06
794	3.403e-05	5.241e-05	3.599e-05	3.248e-06
1000	4.350e-05	6.656e-05	4.591e-05	4.740e-06
1259	5.753e-05	8.831e-05	6.100e-05	5.742e-06
1585	7.611e-05	1.213e-04	8.209e-05	1.025e-05
1995	9.725e-05	1.478e-04	1.043e-04	1.163e-05
2512	1.274e-04	1.885e-04	1.365e-04	1.393e-05
3162	1.691e-04	2.453e-04	1.842e-04	1.751e-05
3981	2.155e-04	2.998e-04	2.317e-04	2.137e-05
5012	2.801e-04	3.811e-04	3.035e-04	2.587e-05
6310	3.721e-04	5.466e-04	4.168e-04	4.562e-05
7943	4.744e-04	6.515e-04	5.185e-04	3.686e-05
10000	6.127e-04	8.034e-04	6.663e-04	3.810e-05
12589	8.228e-04	1.093e-03	8.890e-04	5.185e-05
15849	1.048e-03	1.319e-03	1.125e-03	5.383e-05
19953	1.349e-03	1.680e-03	1.446e-03	7.083e-05
25119	1.795e-03	2.149e-03	1.916e-03	7.464e-05
31623	2.295e-03	2.685e-03	2.434e-03	9.141e-05
39811	2.946e-03	3.404e-03	3.123e-03	1.056e-04
50119	3.934e-03	4.445e-03	4.142e-03	1.154e-04
63096	5.022e-03	5.605e-03	5.263e-03	1.334e-04
79433	6.436e-03	7.357e-03	6.747e-03	1.754e-04

100000	8.546e-03	1.167e-02	8.962e-03	4.378e-04
125893	1.087e-02	1.194e-02	1.126e-02	2.003e-04
158489	1.396e-02	1.521e-02	1.440e-02	2.271e-04
199526	1.843e-02	2.278e-02	1.910e-02	7.139e-04
251189	2.348e-02	2.677e-02	2.417e-02	4.954e-04
316228	3.003e-02	3.441e-02	3.083e-02	6.386e-04
398107	3.943e-02	4.411e-02	4.043e-02	7.953e-04
501187	5.011e-02	5.642e-02	5.134e-02	1.056e-03
630957	6.404e-02	6.994e-02	6.546e-02	9.958e-04

Tab. 4 Tabela de valores para Merge Sort

Gráfico



Img. 4 Merge Sort

Na imagem 4 estão desenhadas as curvas que representam o tempo mínimo, máximo e médio para as experiências feitas com este algoritmo e a regressão linear da curva do tempo médio. Em menos de 60 segundos de tempo médio é possível ordenar arrays com mais de 5,5 x 10^5 elementos.

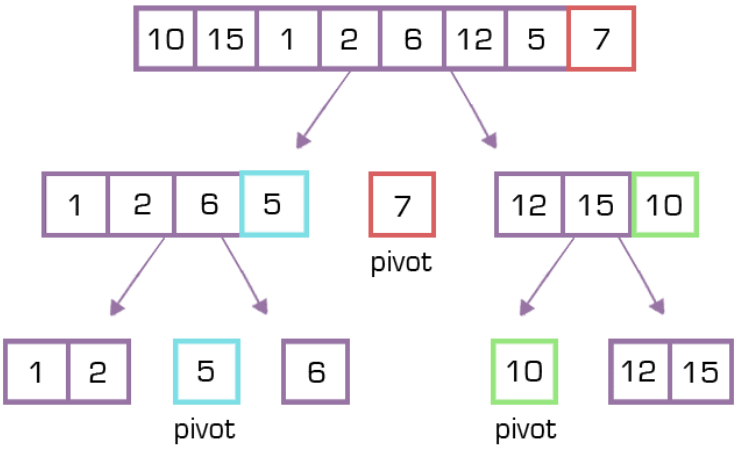
O Merge Sort tem complexidade da ordem *n log n,* tal como se pode ver pela expressão da regressão linear. Tanto o gráfico do tempo máximo como do tempo mínimo acompanham a mesma curva, logo têm complexidades da mesma ordem, mas com coeficientes diferentes.

Quicksort

Algoritmo

É uma rotina de ordenação baseada em Divide and Conquer.
O Quicksort divide o array em três partes tendo em conta um *pivot,* isto é, um elemento do array é escolhido à sorte e é com base nele que o array é dividido. Os valores menores que o *pivot* ficam antes dele, os iguais ficam junto dele e os maiores ficam depois dele. Isto é feito recursivamente, ou seja, após esta primeira passagem, podemos considerar, por exemplo, o valores menores que o pivot e dividir essa parte tendo em conta um novo *pivot* dentro da nova gama de valores e, assim, sucessivamente.
Neste caso, o pior caso é quando o *pivot* escolhido está perto ou é o limite do array, ou seja, se o pivot for o primeiro elemento ou o último. O melhor caso é quando o *pivot* é mais ou menos centrado.

Por exemplo:



Para uma ordem crescente, calhou o pivot ser o último elemento que é 7. Os valores menores que 7 irão ficar atrás dele e os maiores depois dele. De seguida, faz-se a mesma coisa para o "sub-array" dos elementos menores que 7 e dos elementos maiores que 7. O novo pivot no primeiro caso é o 5, os valores menores que 5 ficam atrás dele e os maiores depois dele. No segundo caso, o pivot é 10, coloca-se antes dele os valores menores que 10 e depois dele o valores maiores que 10. Neste caso, o array ficou já ordenado.

Complexidades computacionais

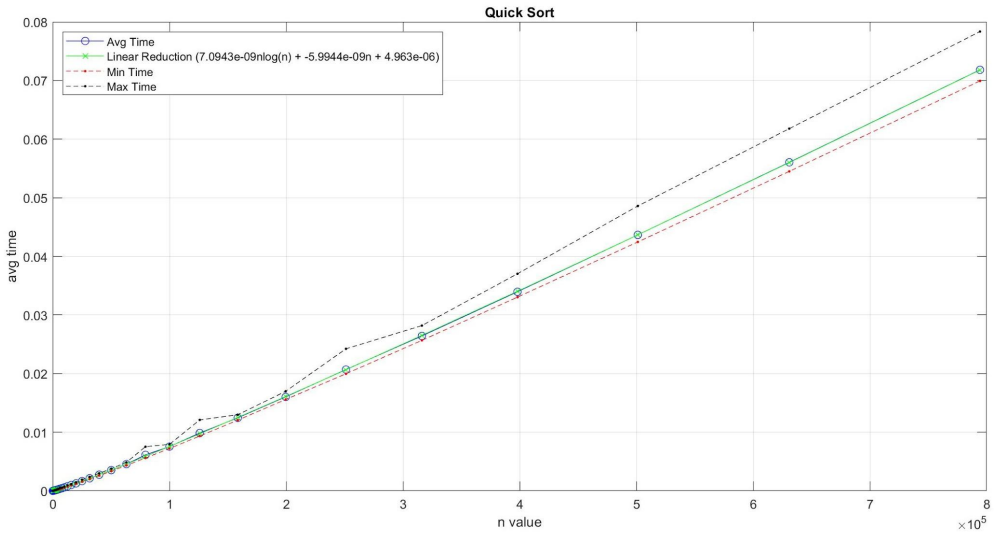
No melhor caso: *O(n log n)*
No caso médio: *O(n log n)*
No pior caso: *O(n²)*

n	min time	max time	avg time	std dev
10	5.730e-07	6.310e-07	5.983e-07	1.281e-08
13	6.110e-07	6.660e-07	6.364e-07	1.311e-08
16	6.520e-07	8.700e-07	6.935e-07	3.608e-08
20	7.590e-07	8.540e-07	8.039e-07	2.228e-08
25	8.500e-07	9.830e-07	9.070e-07	3.059e-08
32	1.004e-06	1.183e-06	1.087e-06	4.242e-08
40	1.207e-06	1.642e-06	1.309e-06	8.249e-08
50	1.450e-06	1.662e-06	1.549e-06	5.182e-08
63	1.798e-06	2.394e-06	1.935e-06	8.510e-08
79	2.254e-06	3.099e-06	2.445e-06	1.544e-07
100	2.885e-06	3.351e-06	3.080e-06	1.018e-07
126	3.725e-06	4.924e-06	3.965e-06	1.609e-07
158	4.794e-06	6.399e-06	5.096e-06	2.142e-07
200	6.249e-06	8.166e-06	6.583e-06	2.510e-07
251	8.105e-06	9.220e-06	8.487e-06	2.094e-07
316	1.084e-05	1.410e-05	1.139e-05	3.920e-07
398	1.429e-05	1.932e-05	1.515e-05	6.804e-07
501	1.797e-05	3.250e-05	1.903e-05	1.774e-06
631	2.355e-05	4.252e-05	2.546e-05	3.695e-06
794	3.161e-05	4.785e-05	3.407e-05	3.714e-06
1000	3.992e-05	6.113e-05	4.222e-05	3.696e-06
1259	5.194e-05	8.610e-05	5.671e-05	7.277e-06
1585	6.768e-05	1.148e-04	7.342e-05	8.922e-06
1995	8.801e-05	1.367e-04	9.460e-05	9.776e-06
2512	1.144e-04	1.721e-04	1.245e-04	1.339e-05
3162	1.488e-04	2.185e-04	1.627e-04	1.742e-05
3981	1.930e-04	2.743e-04	2.105e-04	2.042e-05
5012	2.506e-04	3.483e-04	2.756e-04	2.473e-05
6310	3.264e-04	4.932e-04	3.669e-04	3.903e-05
7943	4.208e-04	5.542e-04	4.611e-04	3.208e-05
10000	5.465e-04	7.317e-04	5.951e-04	3.825e-05
12589	7.099e-04	9.343e-04	7.743e-04	4.396e-05
15849	9.346e-04	1.162e-03	9.987e-04	4.529e-05
19953	1.197e-03	1.460e-03	1.281e-03	5.543e-05

25119	1.550e-03	1.907e-03	1.663e-03	7.411e-05
31623	2.006e-03	2.372e-03	2.141e-03	8.155e-05
39811	2.605e-03	2.988e-03	2.756e-03	8.749e-05
50119	3.352e-03	3.780e-03	3.538e-03	1.018e-04
63096	4.336e-03	4.859e-03	4.560e-03	1.157e-04
79433	5.608e-03	7.532e-03	6.134e-03	4.677e-04
100000	7.220e-03	7.926e-03	7.548e-03	1.563e-04
125893	9.348e-03	1.210e-02	9.861e-03	4.325e-04
158489	1.206e-02	1.298e-02	1.246e-02	1.973e-04
199526	1.559e-02	1.698e-02	1.603e-02	2.460e-04
251189	1.997e-02	2.423e-02	2.069e-02	6.475e-04
316228	2.567e-02	2.819e-02	2.642e-02	4.241e-04
398107	3.304e-02	3.703e-02	3.396e-02	6.264e-04
501187	4.246e-02	4.859e-02	4.367e-02	9.853e-04
630957	5.450e-02	6.179e-02	5.604e-02	1.252e-03
794328	6.993e-02	7.835e-02	7.182e-02	1.428e-03

Tab. 5 Tabela de valores para Quicksort

Gráfico



Img. 5 Quicksort

Na imagem 5 podemos observar o gráfico que apresenta as curvas dos casos mínimo, médio e máximo do QuickSort e a curva da linearização do caso médio (pelo método dos mínimos quadrados). Este método permite ordenar um array de mais de 6,5 x 10^5 elementos em menos de 60 segundos em média.

Este método tem complexidade computacional *n log n* para o melhor caso e caso médio, tal como é possível verificar pela observação do gráfico. Neste, vemos que a regressão linear, também de ordem *n log n,* acompanha as duas linhas referentes ao tempo mínimo e médio. No entanto, para o tempo máximo a curva já não acompanha totalmente a curva da regressão. Ao mesmo tempo, o pior caso deste método seria de ordem n^2, mas ao observar o gráfico vemos que a curva do tempo máximo também não é de ordem n^2. Tal como já foi explicado anteriormente, o script que testa todos estes métodos gera um array pseudo aleatório. O pior caso deste método aconteceria quando o array que se pretendia ordenar estivesse na ordem inversa, mas dificilmente isto se verificaria na prática, muito menos gerando dados pseudo aleatórios.

Rank Sort

Algoritmo

Tanto o Rank Sort como o Selection Sort são usados para números de dados muito pequenos.
Vamos considerar um elemento do array, que é mudado a cada iteração, em que irá comparar-se com cada valor que está no array, contando o número de elementos no array menores do que esse elemento. Essa informação é armazenada no rank desse valor para, posteriormente, ser usado para colocar os elementos nas posições certas sabendo que eram menores do que um determinado valor.
Tal como o Mergesort, necessita de mais espaço em memória.

Complexidades computacionais

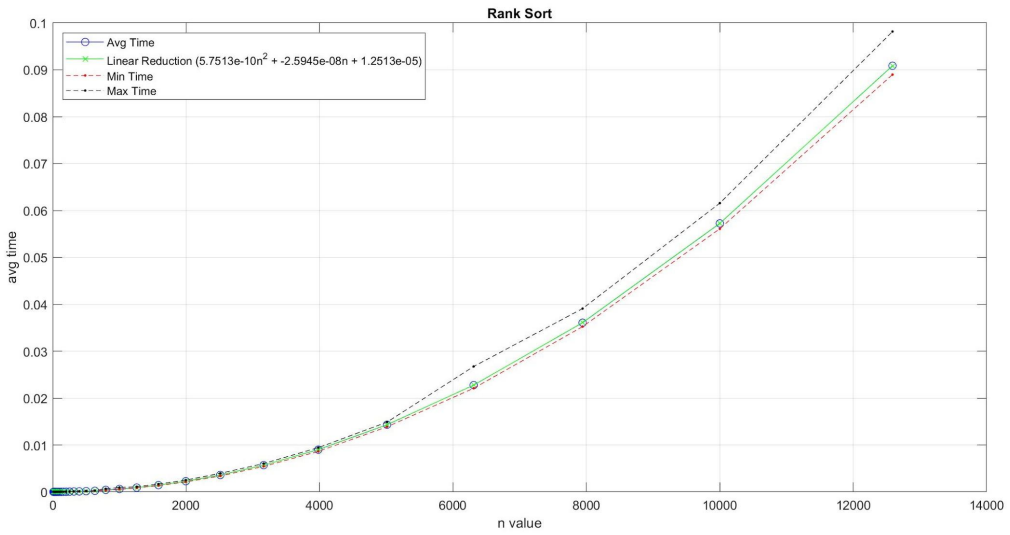
No melhor caso: $O(n^2)$
No caso médio: $O(n^2)$
No pior caso: $O(n^2)$

n	min time	max time	avg time	std dev
10	6.250e-07	6.540e-07	6.339e-07	8.503e-09
13	6.560e-07	7.390e-07	6.742e-07	1.316e-08
16	6.970e-07	7.370e-07	7.115e-07	1.116e-08
20	7.640e-07	8.160e-07	7.833e-07	1.192e-08
25	8.850e-07	9.430e-07	9.099e-07	1.363e-08
32	1.106e-06	1.593e-06	1.161e-06	5.045e-08
40	1.402e-06	1.525e-06	1.455e-06	2.974e-08
50	1.881e-06	2.061e-06	1.959e-06	4.141e-08
63	2.661e-06	3.627e-06	2.789e-06	9.435e-08
79	3.922e-06	4.285e-06	4.089e-06	8.206e-08
100	5.812e-06	8.709e-06	6.213e-06	4.718e-07
126	8.692e-06	1.372e-05	9.284e-06	7.729e-07
158	1.333e-05	1.822e-05	1.394e-05	4.435e-07
200	2.105e-05	5.018e-05	2.551e-05	6.975e-06
251	3.415e-05	7.782e-05	4.339e-05	9.419e-06
316	5.502e-05	1.143e-04	7.334e-05	1.611e-05
398	8.325e-05	1.280e-04	9.052e-05	9.282e-06
501	1.303e-04	2.019e-04	1.441e-04	1.660e-05
631	2.063e-04	2.878e-04	2.246e-04	1.915e-05

794	3.305e-04	6.572e-04	4.214e-04	8.918e-05
1000	5.249e-04	9.366e-04	6.154e-04	9.338e-05
1259	8.393e-04	1.093e-03	9.067e-04	4.966e-05
1585	1.329e-03	1.666e-03	1.429e-03	6.957e-05
1995	2.121e-03	2.509e-03	2.261e-03	8.784e-05
2512	3.387e-03	3.947e-03	3.596e-03	1.235e-04
3162	5.426e-03	6.035e-03	5.691e-03	1.419e-04
3981	8.627e-03	9.428e-03	8.999e-03	1.782e-04
5012	1.387e-02	1.487e-02	1.431e-02	2.047e-04
6310	2.208e-02	2.673e-02	2.276e-02	6.394e-04
7943	3.523e-02	3.908e-02	3.607e-02	6.090e-04
10000	5.609e-02	6.156e-02	5.724e-02	8.554e-04
12589	8.897e-02	9.815e-02	9.086e-02	1.561e-03

Tab. 6 Tabela de valores para Rank Sort

Gráfico



Img. 6 Rank Sort

A imagem 6 apresenta o gráfico com as curvas relativas aos dados do Rank Sort, tempo mínimo, médio, máximo e a regressão linear dos tempos médios de execução. Quer no gráfico, quer na tabela podemos observar a performance deste algoritmo que consegue ordenar arrays com 10 mil elementos em menos de 60 segundos de tempo médio.
Tal como foi dito anteriormente as complexidades computacionais para o caso médio, melhor e pior casos é de ordem n^2. Podemos verificar este facto através da análise do gráfico onde as curvas acompanham a curva da regressão linear que é, também, de ordem n^2.

Selection Sort

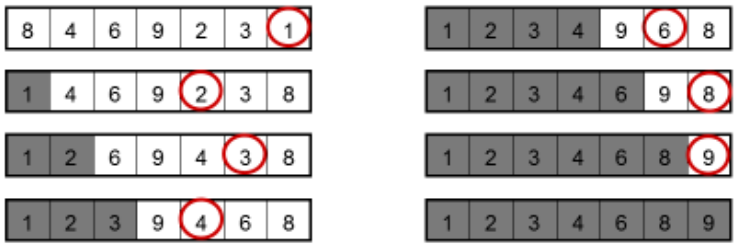
Algoritmo

De modo a simplificar a explicação vamos considerar uma ordenação crescente do array. Este algoritmo consiste em colocar sempre na primeira posição o menor valor do array (seria o maior valor, no caso de ordenação decrescente). Após a primeira passagem por todos os elementos para descobrir o menor valor, sendo que esse fica na posição inicial, apenas se vai considerar o "sub-array" que exclui o elemento já ordenado, ou seja, o primeiro, se o array tiver índices de 0 a n, o "sub-array" conterá os valores dos índices 1 até n. Volta-se a procurar o novo menor valor no sub-array e faz-se exatamente a mesma operação. O "sub-array" vai diminuindo até chegar a ter só um elemento, aí a rotina termina.

Complexidades computacionais

No melhor caso: $O(n^2)$
No caso médio: $O(n^2)$
No pior caso: $O(n^2)$

Por exemplo:



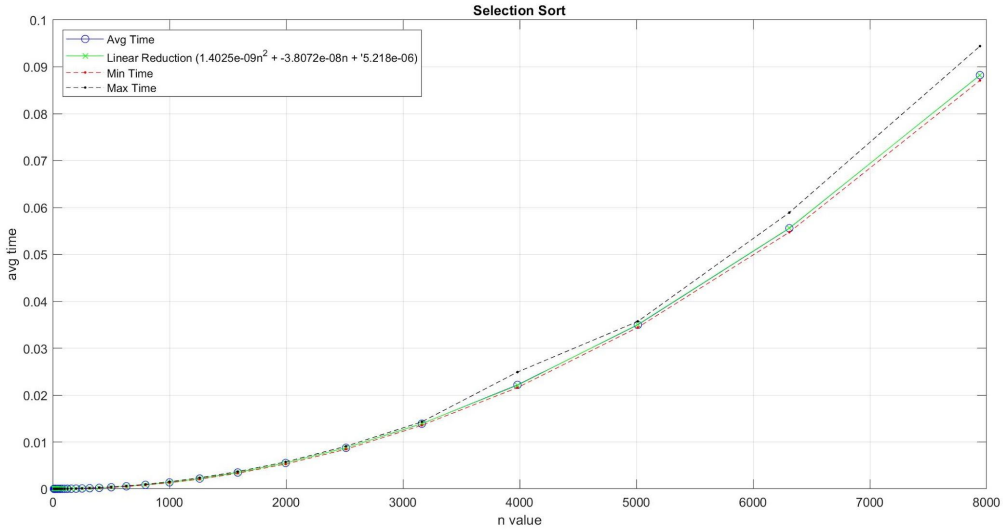
Neste caso, para uma ordenação crescente, procuramos pelo array todo o menor valor e colocamo-lo no início do array. Depois, passamos a considerar os elementos após o 1, "esquecemos" o 1, é como se considerássemos o array [4 6 9 2 3 8]. O menor valor agora é 2 logo passa para o início, de seguida, diminuímos o array retirando o 2, "esquecendo" o 2. O menor valor passa a ser 3, que é colocado no

início do array e o array a considerar passa a ser [9 4 6 8]. Repete-se as mesmas operações até o array estar completamente ordenado.

n	min time	max time	avg time	std dev
10	6.000e-07	6.640e-07	6.336e-07	1.494e-08
13	6.900e-07	7.600e-07	7.253e-07	1.481e-08
16	8.160e-07	9.080e-07	8.566e-07	1.870e-08
20	1.003e-06	1.082e-06	1.038e-06	1.582e-08
25	1.311e-06	1.470e-06	1.351e-06	2.691e-08
32	1.820e-06	1.927e-06	1.859e-06	2.626e-08
40	2.533e-06	2.679e-06	2.595e-06	2.048e-08
50	3.678e-06	4.066e-06	3.787e-06	6.731e-08
63	5.465e-06	5.888e-06	5.653e-06	7.523e-08
79	8.263e-06	8.926e-06	8.553e-06	1.244e-07
100	1.300e-05	1.614e-05	1.369e-05	5.340e-07
126	2.054e-05	3.249e-05	2.147e-05	1.072e-06
158	3.137e-05	4.815e-05	3.296e-05	2.923e-06
200	5.013e-05	8.076e-05	5.274e-05	4.714e-06
251	7.883e-05	1.202e-04	8.435e-05	8.837e-06
316	1.252e-04	1.866e-04	1.347e-04	1.470e-05
398	1.988e-04	2.699e-04	2.139e-04	1.900e-05
501	3.161e-04	4.176e-04	3.431e-04	2.656e-05
631	5.039e-04	6.196e-04	5.432e-04	2.914e-05
794	8.093e-04	9.711e-04	8.630e-04	3.517e-05
1000	1.292e-03	1.519e-03	1.378e-03	4.937e-05
1259	2.063e-03	2.372e-03	2.190e-03	6.812e-05
1585	3.327e-03	3.677e-03	3.480e-03	8.178e-05
1995	5.287e-03	5.749e-03	5.519e-03	1.091e-04
2512	8.444e-03	9.080e-03	8.757e-03	1.472e-04
3162	1.355e-02	1.430e-02	1.389e-02	1.691e-04
3981	2.159e-02	2.490e-02	2.216e-02	4.758e-04
5012	3.442e-02	3.571e-02	3.498e-02	2.930e-04
6310	5.473e-02	5.891e-02	5.557e-02	5.221e-04
7943	8.708e-02	9.440e-02	8.822e-02	9.546e-04

Tab. 7 Tabela de valores para Selection Sort

Gráfico



Img. 7 Selection Sort

As curvas do tempo mínimo, máximo e médio representadas no gráfico da imagem 7 permitem perceber, juntamente com a tabela 7, que o Selection Sort permite ordenar arrays com cerca de 6500 elementos em até 60 segundos de tempo médio.
O gráfico apresenta também uma regressão linear dos dados do tempo médio de ordem n^2. A complexidade computacional do melhor e pior caso e do caso médio são da mesma ordem. Verificamos isso através da observação do gráfico, pois as curvas dos dados de tempo mínimo, máximo e médio são semelhantes à da regressão.

Shaker Sort

Algoritmo

Esta rotina de ordenação ordena um array considerando duas direções. Inicialmente, percorre o array do início para o fim, considerando cada valor no index i e no index i+1, ou seja, o seu adjacente, se estiverem na ordem errada, trocam de posição. Numa ordenação crescente, significa que após a primeira passagem completa, o último valor será o maior; numa ordenação decrescente, será o menor. Assim, o último elemento já está ordenado, na próxima passagem já não terá de ser analisado. Na segunda passagem, a ordenação irá começar de trás para a frente, os primeiros elementos a analisar são n-2 e n-3, e voltam-se a comparar os valores, dois a dois, assim o valor no primeiro index será o menor, numa ordenação crescente ou o maior, se for uma ordenação decrescente. Na terceira iteração, começa-se do início para o fim não sendo necessário comparar o primeiro e o último elementos com os restantes e, assim, sucessivamente.

Por exemplo:
Vamos ordenar de forma crescente o array [2 5 8 1 6].

1. [**2 5** 8 1 6] (2<5, mantêm-se)
2. [2 **5 8** 1 6] (5<8, mantêm-se)
3. [2 5 **8 1** 6] (8>1, trocam-se)
4. [2 5 1 **8 6**] (8>6, trocam-se)
5. [2 5 1 6 8] (fim da 1ª passagem completa, o maior valor está na última posição, por isso, já está ordenado, logo, podemos descartá-lo).
6. [2 5 **1 6**] 8 (1<6, mantêm-se)
7. [2 **5 1** 6] 8 (5>1, trocam-se)
8. [**2 1** 5 6] 8 (2>1, trocam-se)
9. [1 2 5 6] 8 (fim da 2ªa passagem completa, o menor valor está na primeira posição, já está ordenado, logo podemos descartá-lo)
10. 1 [2 5 6] 8 (podemos ver que o array já está ordenado mas o algoritmo irá sempre fazer as passagens todas até ter ordenado os elementos todos)
11. 1 [**2 5** 6] 8 (2<5, mantêm-se)
12. 1 [2 **5 6**] 8 (5<6, mantêm-se)
13. 1 [2 5 6] 8 (fim da 3ªa passagem completa, o maior valor está na última posição logo é descartado)
14. 1 [**2 5**] 6 8 (5>2, mantêm-se e já não há mais elementos a comparar)
15. O array está ordenado

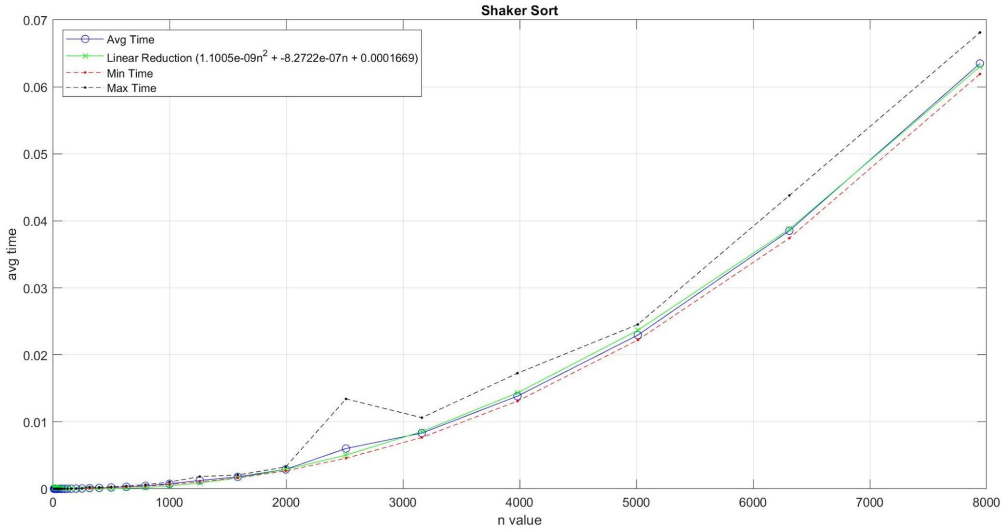
Complexidades computacionais

No melhor caso: $O(n)$
No caso médio: $O(n^2)$
No pior caso: $O(n^2)$

n	min time	max time	avg time	std dev
10	6.390e-07	8.710e-07	7.196e-07	6.660e-08
13	7.180e-07	8.450e-07	7.771e-07	2.988e-08
16	8.120e-07	9.730e-07	8.903e-07	3.816e-08
20	9.760e-07	1.248e-06	1.087e-06	5.733e-08
25	1.206e-06	1.546e-06	1.358e-06	7.499e-08
32	1.630e-06	2.056e-06	1.830e-06	1.009e-07
40	2.208e-06	3.046e-06	2.536e-06	1.819e-07
50	3.063e-06	3.955e-06	3.444e-06	1.942e-07
63	4.498e-06	6.351e-06	5.210e-06	4.370e-07
79	6.450e-06	9.276e-06	7.262e-06	5.714e-07
100	9.681e-06	1.446e-05	1.163e-05	1.268e-06
126	1.464e-05	2.239e-05	1.719e-05	1.772e-06
158	2.181e-05	4.075e-05	2.505e-05	3.039e-06
200	3.302e-05	5.923e-05	3.747e-05	4.843e-06
251	4.976e-05	8.729e-05	5.684e-05	7.508e-06
316	7.651e-05	2.735e-04	1.049e-04	3.223e-05
398	1.153e-04	2.289e-04	1.396e-04	2.468e-05
501	1.744e-04	3.004e-04	2.044e-04	3.042e-05
631	2.656e-04	4.237e-04	3.061e-04	3.790e-05
794	4.058e-04	6.262e-04	4.657e-04	5.090e-05
1000	6.461e-04	1.010e-03	7.547e-04	8.304e-05
1259	1.048e-03	1.834e-03	1.253e-03	1.887e-04
1585	1.628e-03	2.065e-03	1.781e-03	1.062e-04
1995	2.685e-03	3.265e-03	2.904e-03	1.356e-04
2512	4.571e-03	1.343e-02	6.017e-03	1.690e-03
3162	7.669e-03	1.062e-02	8.328e-03	5.994e-04
3981	1.309e-02	1.726e-02	1.384e-02	8.137e-04
5012	2.221e-02	2.454e-02	2.295e-02	4.201e-04
6310	3.742e-02	4.379e-02	3.853e-02	9.627e-04
7943	6.192e-02	6.812e-02	6.349e-02	1.094e-03

Tab. 8 Tabela de valores para Shaker Sort

Gráfico



Img. 8 Shaker Sort

A imagem 8 apresenta o gráfico referente ao algoritmo de ordenação Shaker Sort. Este contém as curvas referentes aos dados do tempo médio, máximo e mínimo de execução para vários valores de n. Tal como podemos perceber através da tabela 8, este método ordena arrays com mais de 7500 elementos em menos de 60 segundos de tempo médio.

O gráfico apresenta ainda a regressão linear dos valores médios do tempo de execução do algoritmo. A expressão da regressão é de ordem n^2, e a sua curva é semelhante às curvas dos tempos mínimo, médio e máximo, logo podemos concluir que a expressão dessas curvas será da mesma ordem. No entanto, a complexidade computacional para o melhor caso deste método é de ordem *n.* Como já foi explicado anteriormente, o facto de os arrays serem pseudo aleatórios e, também, a limitação do número de experiências levam a que o melhor caso teórico não se verifique porque o array considerado dificilmente estará ordenado por defeito.

Shell Sort

Algoritmo

O Shell Sort é semelhante ao Insertion Sort. Neste caso, pretendemos ordenar o array considerando posições do array distanciadas num determinado valor (seja este valor representado h). Repare-se que se considerarmos uma ordenação com h=1 temos exatamente o Insertion Sort. O que irá acontecer, portanto, é que o valor na posição i, será comparado com o valor na posição i+h, que por sua vez é comparado com o valor na posição i+2h, etc. Vamos comparar índices na mesma progressão aritmética. O h entretanto vai diminuindo até chegar a 1. Isto é feito sequencialmente até o array estar completamente ordenado. Isto permite colocar os elementos mais rapidamente numa posição mais próxima de onde deve realmente ficar.

Por exemplo:
Queremos ordenar os seguintes números por ordem crescente considerando h=4 inicialmente. Vamos comparar os valores distanciados em h posições, se o primeiro por maior que o segundo, trocam, senão, mantém-se iguais.
7 6 8 9 **3** 2 10 5 1
3 **6** 8 9 7 **2** 10 5 1
3 2 **8** 9 7 6 **10** 5 1
3 2 8 **9** 7 6 10 **5** 1
3 2 8 5 7 6 10 9 1
Se diminuirmos h para h=3
3 2 8 **5** 7 6 10 9 1
3 **2** 8 5 **7** 6 10 9 1
3 2 **8** 5 7 **6** 10 9 1
3 2 6 5 7 8 10 9 1
Agora, para h=2
3 2 **6** 5 7 8 10 9 1
3 **2** 6 **5** 7 8 10 9 1
3 2 6 5 **7** 8 **10** 9 1
3 2 6 5 7 **8** 10 **9** 1
Finalmente, para h=1 (igual ao insertion sort, ver insertion sort para explicação detalhada)
3 **2** 6 5 7 8 10 9 1
2 3 **6** 5 7 8 10 9 1
2 3 6 **5** 7 8 10 9 1
2 3 5 6 **7** 8 10 9 1

2 3 5 6 7 **8** 10 9 1
2 3 5 6 7 8 **10** 9 1
2 3 5 6 7 8 10 **9** 1
2 3 5 6 7 8 9 10 **1**
1 2 3 5 6 7 8 9 10 -> Ordenação completa

Complexidades computacionais

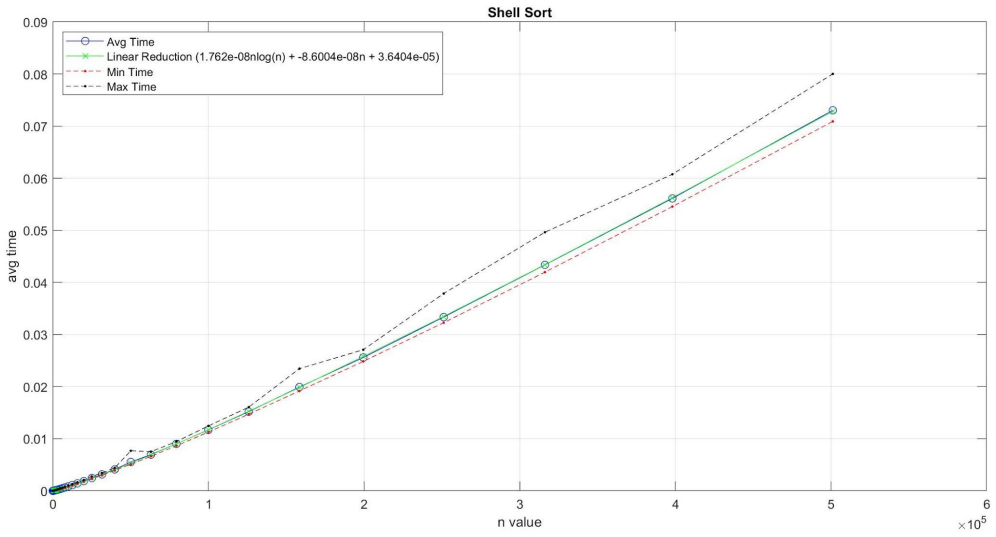
Relativo pois depende da distância h (gap) utilizada. O melhor caso conhecido é *O(n log2 n)* e o pior caso conhecido é *O(n)* total, *O(1)* auxiliar.

n	min time	max time	avg time	std dev
10	6.190e-07	6.960e-07	6.550e-07	1.819e-08
13	6.720e-07	9.420e-07	7.374e-07	5.212e-08
16	7.330e-07	8.650e-07	7.897e-07	2.941e-08
20	8.300e-07	9.800e-07	8.975e-07	3.386e-08
25	9.590e-07	1.130e-06	1.039e-06	4.076e-08
32	1.154e-06	1.355e-06	1.246e-06	4.715e-08
40	1.397e-06	1.819e-06	1.515e-06	7.459e-08
50	1.731e-06	2.024e-06	1.862e-06	6.775e-08
63	2.188e-06	2.588e-06	2.351e-06	8.426e-08
79	2.781e-06	3.459e-06	2.966e-06	1.078e-07
100	3.572e-06	3.997e-06	3.784e-06	1.006e-07
126	4.658e-06	5.868e-06	4.917e-06	1.604e-07
158	6.113e-06	7.617e-06	6.424e-06	1.858e-07
200	8.118e-06	1.059e-05	8.508e-06	3.424e-07
251	1.063e-05	1.177e-05	1.115e-05	2.499e-07
316	1.379e-05	2.702e-05	1.449e-05	1.087e-06
398	1.832e-05	3.171e-05	1.942e-05	1.558e-06
501	2.402e-05	3.754e-05	2.493e-05	1.603e-06
631	3.159e-05	5.045e-05	3.337e-05	3.131e-06
794	4.113e-05	6.742e-05	4.388e-05	5.147e-06
1000	5.381e-05	8.226e-05	5.707e-05	5.865e-06
1259	7.068e-05	1.126e-04	7.621e-05	9.018e-06
1585	9.255e-05	1.389e-04	9.954e-05	1.073e-05
1995	1.210e-04	1.765e-04	1.308e-04	1.351e-05
2512	1.577e-04	2.262e-04	1.725e-04	1.703e-05
3162	2.050e-04	2.916e-04	2.251e-04	2.272e-05
3981	2.680e-04	3.674e-04	2.896e-04	2.428e-05
5012	3.515e-04	4.832e-04	3.831e-04	3.106e-05
6310	4.569e-04	5.957e-04	5.001e-04	3.097e-05

7943	5.982e-04	8.068e-04	6.544e-04	4.014e-05
10000	7.890e-04	1.017e-03	8.524e-04	4.273e-05
12589	1.028e-03	1.288e-03	1.103e-03	5.128e-05
15849	1.342e-03	1.638e-03	1.440e-03	6.537e-05
19953	1.750e-03	2.087e-03	1.867e-03	7.314e-05
25119	2.296e-03	2.751e-03	2.441e-03	9.571e-05
31623	2.985e-03	3.434e-03	3.161e-03	1.040e-04
39811	3.907e-03	4.401e-03	4.115e-03	1.167e-04
50119	5.104e-03	7.688e-03	5.542e-03	4.842e-04
63096	6.639e-03	7.511e-03	6.952e-03	1.733e-04
79433	8.638e-03	9.447e-03	9.010e-03	1.838e-04
100000	1.128e-02	1.246e-02	1.170e-02	2.223e-04
125893	1.471e-02	1.606e-02	1.519e-02	2.565e-04
158489	1.917e-02	2.345e-02	1.992e-02	7.167e-04
199526	2.485e-02	2.708e-02	2.558e-02	4.247e-04
251189	3.227e-02	3.787e-02	3.335e-02	8.737e-04
316228	4.197e-02	4.963e-02	4.337e-02	1.170e-03
398107	5.455e-02	6.076e-02	5.612e-02	1.061e-03
501187	7.090e-02	8.002e-02	7.303e-02	1.588e-03

Tab. 9 Tabela de valores para Shell Sort

Gráfico

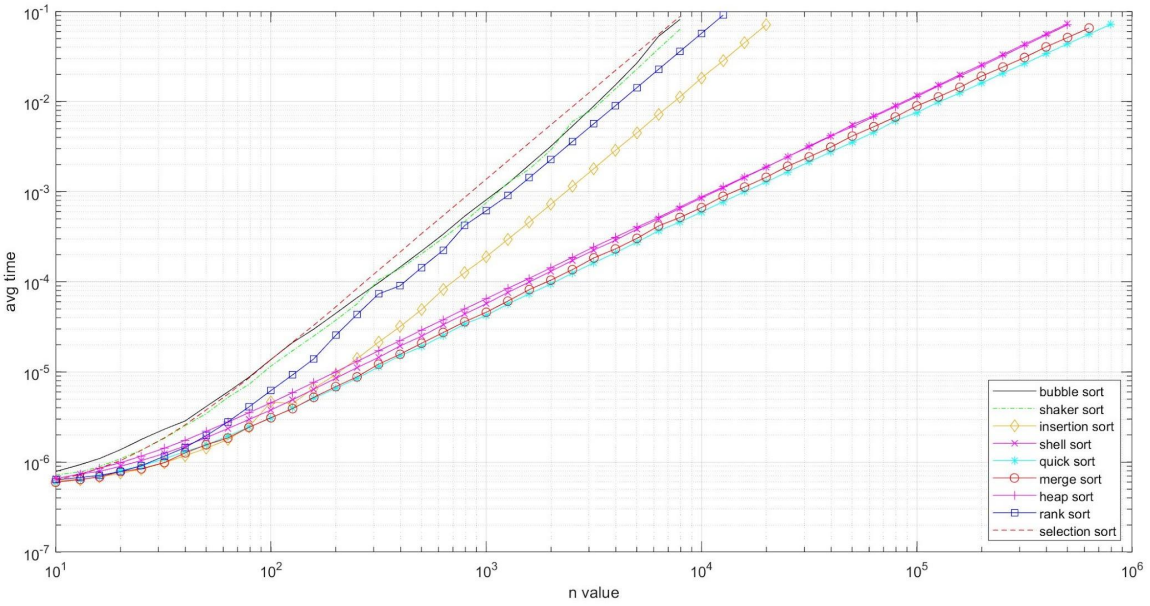


Img. 9 Shell Sort

A imagem 9 apresenta o gráfico com as curvas dos tempos mínimo, médio e máximo do método Shell Sort. Este algoritmo permite ordenar array com dimensão superior a 400 mil elementos em menos de 60 segundos de tempo médio, como é possível verificar na tabela e no gráfico.

Não podemos tirar muitas conclusões acerca da relação entre a regressão linear dos tempos médios de execução e a sua complexidade computacional uma vez que esta é relativa ao h.

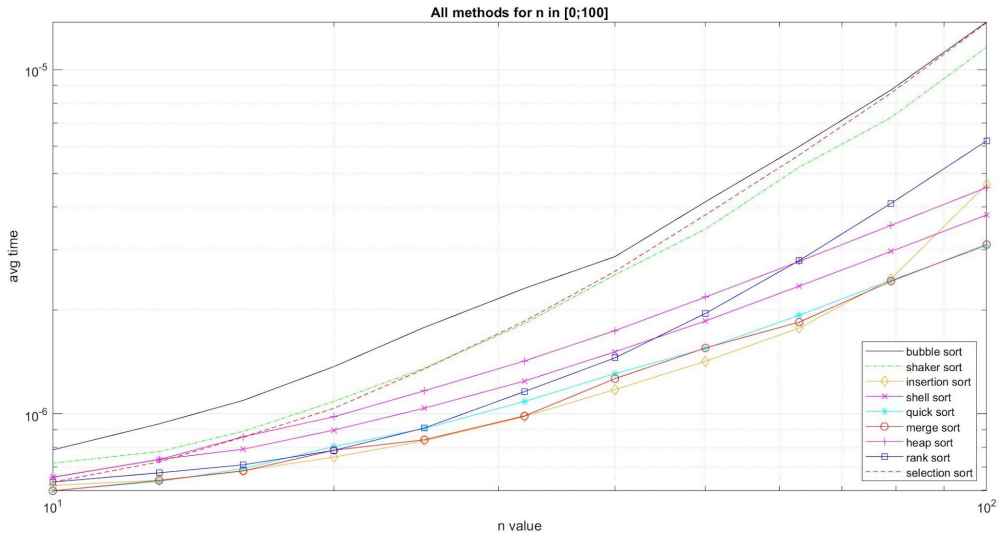
Comparação dos Métodos



Img. 10 Todos os métodos sort

A imagem 10 apresenta o tempo médio de cada um dos métodos Sort apresentados neste trabalho. Mostrando todos no mesmo gráfico conseguimos ter uma melhor perspetiva da eficácia de cada um deles e tirar conclusões. De seguida iremos falar dos métodos mais eficientes para intervalos específicos de *n* mas por aqui concluímos pela observação do gráfico que o método mais eficiente à medida que *n* cresce para infinito é o método quicksort que permite ordenar um array de mais de 6,5 x 10^5 elementos em menos de 60 segundos em média.

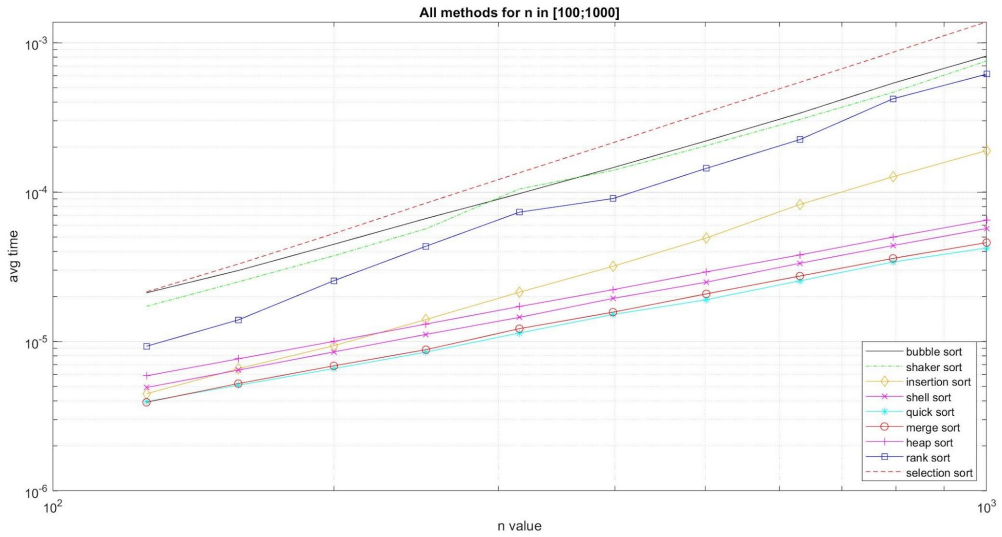
Para n em [0;100]



Img. 11 Tempo médio de execução para n=[0,100]

Pelo estudo da imagem 11 concluímos que no início do gráfico até cerca de n=50 os métodos mais eficientes são o merge sort e o insertion sort. De n=50 até n=100 o insertion sort e o rank sort tornam-se menos eficientes comparados ao merge sort e ao quicksort que mantém o seu crescimento constante do início ao fim. Em contrapartida, vemos métodos como bubble sort e selection sort que demoram bastante mais tempo para chegar aos mesmos valores de *n* que os outros métodos.

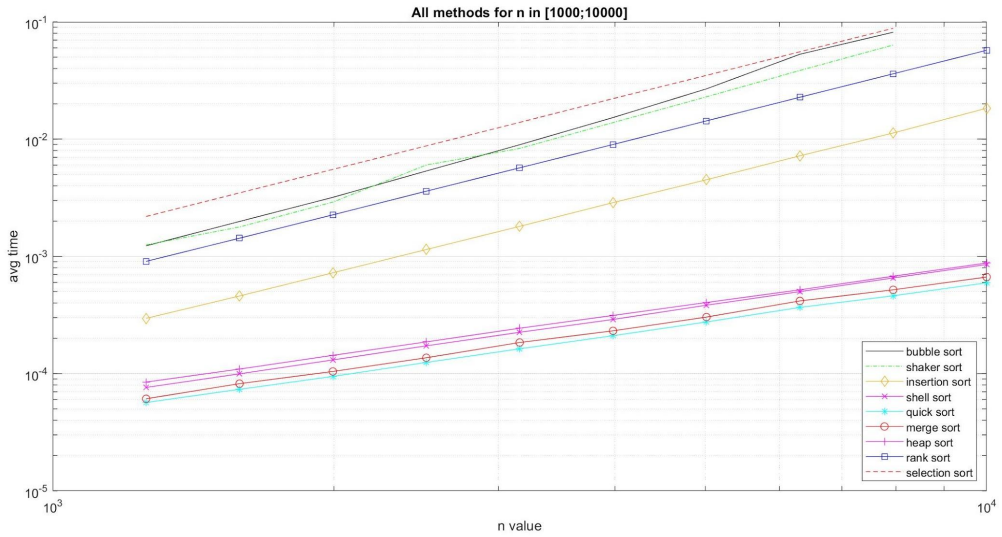
Para n em [100;1000]



Img. 12 Tempo médio de execução para n=[100,1000]

Pelo estudo da imagem 12 vemos que métodos como quick sort e merge sort continuam a ser os mais eficientes e de seguida, mantendo também um crescimento constante temos o shell sort e o heap sort. Mais uma vez o insertion sort começa com uma boa eficácia nos primeiros valores de n mas à medida que este cresce a eficácia diminui bastante. Por fim vemos métodos como o selection sort que continua sendo o método menos eficaz quanto maior for o valor de *n*.

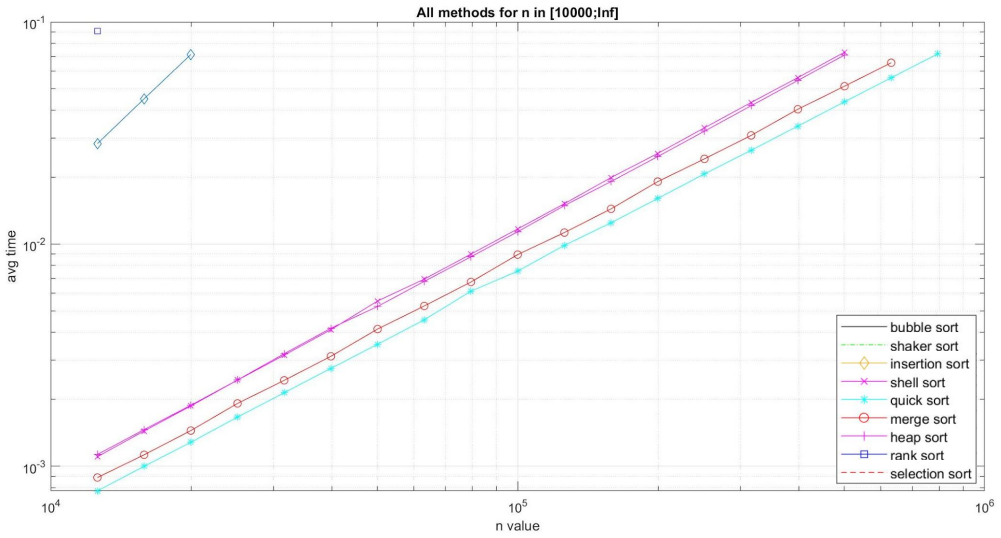
Para n=[1000;10000]



Img. 13 Tempo médio de execução para n=[1000,10000]

Na imagem 13 é-nos apresentado um gráfico onde vemos claramente 2 partes distintas. Temos um grupo que continua com grande eficácia para estes valores de *n* e outros que se tornam bastante inúteis nestes casos. Mantêm-se a liderança do quicksort seguido do merge sort, shell sort e heapsort, todos se mostram opções válidas para problemas desta granditude, mas no lado oposto, o tempo médio de métodos como selection sort, shaker sort, bubble sort e rank sort, apesar de não terem um crescimento muito elevado comparado aos valores de *n* do gráfico anterior continuam com um tempo médio de execução bastante superior aos métodos apresentados anteriormente.

Para n=[10000;inf]



Img. 14 Tempo médio de execução para n=[10000,inf]

Vemos que para valores enormes de n=10000 até ao infinito há métodos que se tornam tão ineficientes que nem chegam a estes valores no intervalo de tempo especificado para este problema. Também vemos que o merge sort e o bubble sort apesar de, muito ineficientes, ainda conseguem atingir valores de *n* um pouco maiores que 10000 mas não passam daí.
Concluímos deste gráfico e com conhecimento dos anteriores que os métodos que inicialmente eram considerados os mais eficazes mantiveram um crescimento constante do seu tempo de execução para o aumento de *n* e tanto para n=100 como n a crescer para o infinito os melhores métodos sort permanece o quick sort, seguido do merge sort e o shell sort e heap sort, que se tornam bastante semelhantes como é observável.

Número de linhas de código ocupadas por cada método

Bubble Sort	19
Heap Sort	39
Insertion Sort	12
Merge Sort	26
Quick Sort	70
Rank Sort	19
Selection Sort	17
Shaker Sort	30
Shell Sort	18

Tab. 10 Número de linhas de código necessárias para implementar cada método

Conclusão

O algoritmo mais consistente e mais eficiente é o Quick Sort. No entanto, este algoritmo ocupa 70 linhas de código e a sua implementação não é muito simples.
O melhor algoritmo depende da situação. Se o objetivo for ordenar arrays até 100 elementos a melhor opção é o Insertion Sort, é rápido e é de implementação simples ocupando apenas 12 linhas de código. Para arrays maiores temos de escolher entre o Merge Sort e o Quick Sort. O Merge Sort será uma boa opção para casos em que a eficiência do código não tem de ser máxima. O que não quer dizer que o Merge Sort seja ineficiente, apenas não é o mais rápido, mas ocupa apenas 26 linhas enquanto que o Quick Sort ocupa 70, este pode ser um fator decisivo na escolha de um em deferimento do outro. No caso de se querer atingir o menor tempo de execução possível deve-se optar pelo Quick Sort.

Caraterísticas do PC

- HP EliteBook 840 G3
- Intel(R) Core(™) i7-6500U CPU @ 2.50GHz 2.59GHz
- 16.0GB de RAM

Foi usada uma máquina virtual através da Oracle VM VirtualBox com o Linux Ubuntu instalado. Esta máquina tem as seguintes características:

- 8GB de RAM
- 2 dos 4 processadores lógicos do host

Conclusão

Os dados foram recolhidos pelo grupo com um tempo de execução limite de 60 segundos, se esse valor fosse mais alto a quantidade de dados seria maior, mas não iria afetar os resultados finais. A eficiência de cada algoritmo foi analisada tendo em conta os tempos médios e com recurso a regressões lineares. Logo, esse fator, embora tivesse impacto nos resultados, não refuta as conclusões apresentadas anteriormente, porque a taxa de crescimento dos tempos médios de execução seria semelhante à apresentada.

Com este trabalho, foi possível perceber que os valores do melhor e do pior caso dificilmente atingem por terem uma possibilidade muito remota de acontecer. No entanto, conseguimos perceber o impacto da complexidade computacional de um algoritmo no seu tempo de execução à medida que o *n* vai aumentando. Assim como, a perceber que mesmo com complexidades computacionais da mesma ordem, dois algoritmos podem não ter a mesma eficiência como por exemplo o Insertion Sort e o Bubble Sort.

Em suma, se pretendermos escolher um algoritmo, não só de ordenação, devemos escolher o que mais se adequa à nossa necessidade e não necessariamente o que tem melhor desempenho em termos de tempo. Por vezes, a complexidade de um algoritmo não justifica o seu uso, mesmo que seja mais eficiente. O uso destes algoritmos pode até ser contraproducente, pois a sua complexidade mais elevada levará a que seja mais difícil transferir o projeto para outra pessoa ou que a sua manutenção seja mais demorada. É sempre necessário analisar qual é o verdadeiro objetivo do nosso código e fazer os compromissos necessários para atingir uma solução robusta, eficaz e eficiente.

Bibliografia

> https://www.geeksforgeeks.org/bubble-sort/
> https://www.geeksforgeeks.org/heap-sort/
> https://www.geeksforgeeks.org/binary-heap/
> https://www.geeksforgeeks.org/insertion-sort/
> https://www.geeksforgeeks.org/merge-sort/
> https://www.geeksforgeeks.org/quick-sort/
> http://alecu.ase.ro/articles/ie_en_2005.pdf
> https://www.geeksforgeeks.org/selection-sort/