ALGORITMOS DE ORDENAÇÃO

Algoritmos e Estruturas de Dados (40437)

2020/2021

Trabalho realizado por:

Tiago Alvim 95584 49%

Vasco Costa 97746 51%

Introdução

A ordenação de informação foi desde sempre um assunto muito relevante na computação sendo uma das suas tarefas primordiais. Foi algo que sempre apelou a muitos investigadores uma vez que é uma matéria complexa e interessante. Sendo que é também uma temática muito importante e indispensável nos dias em que nos encontramos.

Existem diferentes meios de classificação que são relevantes consoante o meio em que se pretende utilizar, a complexidade computacional do mesmo, em relação ao melhor, pior caso e à situação média. Havendo algoritmos cuja complexidade varia consoante a situação e outros que são garantidamente sempre da mesma forma, tanto no melhor como no pior caso, esta situação é relevante quando se pretende estabilidade de computação, o que garante segurança pois não é preciso ter em conta qualquer variância. No entanto, existem algoritmos que necessitam de estruturas secundárias de dados o que torna necessário ter em atenção quando se pretendem arrumar grandes volumes de dados, dado que irá relacionar-se diretamente com o *hardware* utilizado uma vez que poderá não haver espaço suficiente ou até perder muito desempenho quando não é possível encaixar-se tudo na cache do processador ou em *RAM*, levando a uma velocidade de acesso muito mais lenta o que se irá notar no desempenho. Também poderá ser relevante se o algoritmo apresentado é estável ou instável, o que se torna importante quando a organização consecutiva por parâmetros é necessária.

Nos dias de hoje com a possibilidade de se terem mais *cores* de forma acessível, a paralelização de computação é cada vez mais relevante havendo algoritmos que conseguem tirar proveito do mesmo tornando-se excelentes. Sendo que no espaço empresarial, servidores, a utilização de placas gráficas permite ainda maior paralelização, aumentando assim o desempenho e reduzindo custos.

Métodos extra

Após termos procurado alguns métodos que aproveitassem mais que um *thread* no computador, não conseguimos implementar nenhum devido à nossa falta de conhecimento na área. No entanto, encontrámos um código bastante interessante[1] que acreditamos que bem implementado seria de extrema relevância.

Bogo_sort

Estando curiosos como seria um método muito mau decidimos dar uma abordagem a este método em que a ideia é muito simples, se a lista estiver ordenada então já está feito, se não estiver, reordena-se à sorte até que esteja. Assim sendo, este é um método muito ineficiente pior ainda que *bubble_sort* e qualquer outro método de n^2 sendo que poderá ser interessante fazer-se o estudo para o caso médio, como veremos mais à frente.

Tree_sort

Decidimos também implementar esta estratégia adicional devido a julgarmos que esta era interessante e apesar da necessidade de se recorrer a uma estrutura extra, ou seja, maior uso de memória, se usada como base no armazenamento de dados permite uma busca e uma inserção muito eficiente. Este método como o nome indica consiste em usar-se uma árvore binária, optou-se por escolher o primeiro número do *array* desorganizado para ser a raiz, mas servindo qualquer outro elemento, os que são menores vão para a esquerda do nó e se forem maiores serão colocados à direita, após fazerem-se as comparações e encontrado um lugar

[1] https://github.com/karuto/Parallel-Sample-Sort/blob/master/main.c

vazio cria-se um nó nesse local com esse valor. Este método permite então inserção em ordem $log(n)$ na melhor situação. E tendo n elementos obtém-se $nlog(n)$ para a criação e ordenação. Posteriormente é feito mais um processo de ordem n com o intuito de se ler a árvore na totalidade e de se colocar no *array* inicial os dados ordenados, mas como ordem n é menor que ordem $n\ log(n)$ este torna-se irrelevante segundo a notação assintótica.

Análise[2]

Tabela 1

Algoritmo	Melhor	Médio	Pior	Comentários
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	
Shaker sort	$O(n)$	$O(n^2)$	$O(n^2)$	
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	
Shell sort	?	?	?	
Quicksort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n^2)$	
Mergesort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n\ log\ n)$	Requer extra espaço
Heapsort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n\ log\ n)$	
Rank sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Requer extra espaço
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Tree sort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n^2)$	Requer extra espaço
Bogo sort	$O(n)$	$O(n!)$	$O(\infty)$	

Excluindo os últimos dois métodos a justificação encontra-se no material de aula fornecido. Para *tree_sort* afirmamos que este apresenta ordem $n\ log\ n$ no melhor e médio caso como explicado anteriormente. E para o pior caso justificamos que é n^2 pois se a lista já estiver ordenada leva à criação de uma árvore não balanceada na nossa versão, pelo que a inserção será feita em ordem n. Uma *AVL tree* não iria levar a este problema, no entanto a sua implementação é mais complexa e para uma ocorrência tão rara poderá não se justificar.

Para *bogo_sort* apresentamos como melhor situação este já estar ordenado em que temos ordem n mas caso não esteja, então uma permutação terá que ser calculada e com o método que se implementou poderá ainda ser pior pois existe sempre a possibilidade de se repetir uma anterior. Este método é pendente maioritariamente da sorte de acertar exatamente na permutação em que temos todos os números ordenados e como tal pode nunca acontecer pelo que o pior caso vai tender para infinito.

Para a obtenção dos dados primeiramente verificámos os nossos métodos compilando e usando *-test* e posteriormente realizámos com a opção *-measure*. Para se tentar descobrir as constantes associadas aos polinómios que poderão descrever o tempo em função do número de elementos utilizou-se o *matlab* com o comando *cftool*. Nesta ferramenta escolheu-se quando se esperava um polinómio quadrado a opção polinomial e ordem quadrada tendo a robustez desligada. Quando nos deparámos na situação em que se esperava ordem $n\ log\ n$ optou-se pela opção de equação à medida, introduzindo-se a forma do polinómio que se esperava $a * n * \log(b * n) + cn + d$. Podendo-se erradamente não ter alterado as

[2] Os dados foram obtidos em *Intel® Core(TM) i7-4720HQ 2.60GHz* com *Turbo Boost ativado* com *12Gb* de *RAM* a *1600MHz*

predefinições em vez da escolha do algoritmo de *Levenberg-Marquardt* e porventura a opção da robustez desligada.

Bubble_sort e *shaker_sort*

Devido a estes dois métodos terem uma natureza muito semelhante sendo que o segundo tem apenas o acréscimo de se fazerem passagens no sentindo inverso, achou-se pertinente comparar estes dois e de que maneira um é mais eficiente que o outro.

Tabela 2

	Bubble sort			Shaker sort		
n	min time	max time	avg time	min time	max time	avg time
10	1,00E-07	2,00E-07	1,61E-07	1,00E-07	2,00E-07	1,58E-07
13	2,00E-07	3,00E-07	2,55E-07	2,00E-07	3,00E-07	2,23E-07
16	3,00E-07	5,00E-07	3,84E-07	2,00E-07	4,00E-07	3,15E-07
20	5,00E-07	7,00E-07	5,84E-07	4,00E-07	5,00E-07	4,53E-07
25	7,00E-07	1,00E-06	8,82E-07	6,00E-07	8,00E-07	6,94E-07
32	1,20E-06	1,50E-06	1,37E-06	9,00E-07	1,20E-06	1,07E-06
40	1,80E-06	2,30E-06	2,06E-06	1,40E-06	1,80E-06	1,61E-06
50	2,80E-06	3,40E-06	3,07E-06	2,10E-06	2,70E-06	2,41E-06
63	4,30E-06	5,10E-06	4,70E-06	3,30E-06	4,10E-06	3,67E-06
79	6,50E-06	7,70E-06	7,11E-06	5,00E-06	6,00E-06	5,50E-06
100	1,01E-05	1,18E-05	1,09E-05	7,70E-06	9,10E-06	8,39E-06
126	1,51E-05	1,76E-05	1,64E-05	1,17E-05	1,36E-05	1,26E-05
158	2,32E-05	2,64E-05	2,48E-05	1,77E-05	2,03E-05	1,89E-05
200	3,43E-05	3,86E-05	3,63E-05	2,69E-05	3,05E-05	2,85E-05
251	5,02E-05	5,51E-05	5,25E-05	4,01E-05	4,59E-05	4,24E-05
316	7,52E-05	8,63E-05	7,78E-05	6,04E-05	6,70E-05	6,36E-05
398	1,13E-04	1,24E-04	1,16E-04	9,15E-05	1,00E-04	9,53E-05
501	1,71E-04	1,87E-04	1,76E-04	1,39E-04	1,50E-04	1,44E-04
631	2,63E-04	2,81E-04	2,72E-04	2,11E-04	2,57E-04	2,18E-04
794	3,98E-04	4,38E-04	4,12E-04	3,21E-04	3,91E-04	3,33E-04
1000	6,14E-04	6,82E-04	6,34E-04	4,95E-04	5,80E-04	5,13E-04
1259	9,39E-04	1,07E-03	9,83E-04	7,69E-04	8,59E-04	8,03E-04
1585	1,47E-03	1,72E-03	1,54E-03	1,20E-03	1,33E-03	1,24E-03
1995	2,34E-03	2,72E-03	2,45E-03	1,89E-03	2,04E-03	1,95E-03
2512	3,83E-03	4,35E-03	3,99E-03	3,02E-03	3,23E-03	3,11E-03
3162	6,48E-03	7,13E-03	6,68E-03	4,92E-03	5,33E-03	5,03E-03
3981	1,13E-02	1,25E-02	1,17E-02	8,25E-03	8,71E-03	8,42E-03
5012	2,02E-02	2,18E-02	2,07E-02	1,41E-02	1,52E-02	1,44E-02
6310	3,56E-02	3,78E-02	3,63E-02	2,46E-02	2,80E-02	2,51E-02
7943	6,18E-02	6,51E-02	6,28E-02	4,28E-02	4,55E-02	4,37E-02
10000	1,06E-01	1,09E-01	1,07E-01	7,39E-02	7,71E-02	7,51E-02
12589	ND	ND	ND	1,26E-01	1,72E-01	1,28E-01

Para *bubble_sort* conseguiu-se uma aproximação polinomial a $1.261 * 10^{-9}n^2 - 2.075 * 10^{-6}n + 4.885 * 10^{-4}$ e para *shaker_sort* um polinómio de $9.415 * 10^{-10}n^2 - 1.847 * 10^{-6}n + 5.312 * 10^{-4}$. Deste modo podemos esperar que *shaker_sort* cresça mais lentamente, apresentam uma ordem de grandeza de diferença no termo de maior grau que para valores de n mais elevados se deverá notar ainda mais.

Como podemos verificar graficamente *bubble_sort* é relativamente mais lento que o outro método. Não se esperava que *shaker_sort* tivesse uma melhoria tão percetível pois a única diferença é ir colocando o elemento mais pequeno no início. Em que se reduz o tamanho do *array* alternadamente em cada um dos lados. Este método apresenta também uma implementação mais complexa por essa mesma passagem no sentido inverso.

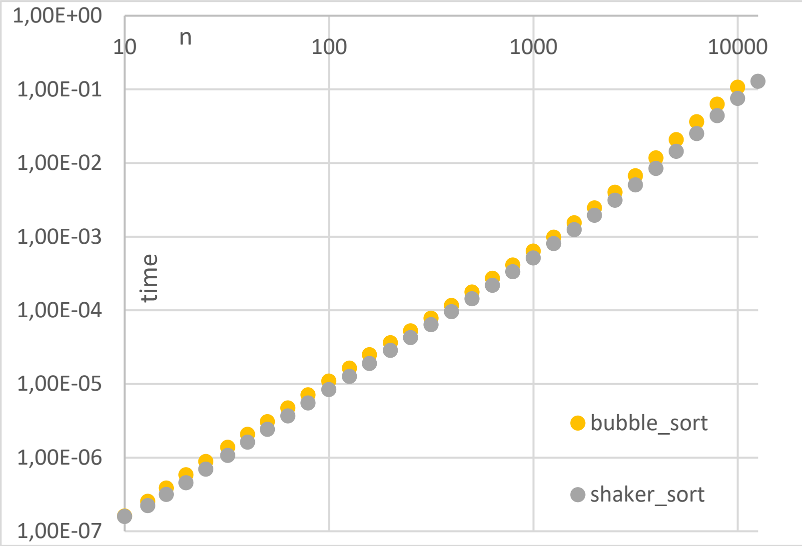

Gráfico 1

Insertion_sort e *Shell_sort*

Dado que *Shell_sort* é uma implementação de *insertion_sort* mas com uma variação que passa por em vez de se fazerem comparações com elementos adjacentes fazem-se com elementos mais separados. Necessitando a realização de cálculos prévios para a escolha do h aumentando as linhas de código e complicando a sua interpretação, mas como se pode verificar este é um método bem melhor.

Tabela 3

	Insertion sort			Shell sort		
n	min time	max time	avg time	min time	max time	avg time
10	0,00E+00	1,00E-07	9,47E-08	1,00E-07	2,00E-07	1,44E-07
13	1,00E-07	2,00E-07	1,14E-07	1,00E-07	3,00E-07	1,95E-07
16	1,00E-07	2,00E-07	1,66E-07	2,00E-07	3,00E-07	2,70E-07
20	2,00E-07	3,00E-07	2,06E-07	3,00E-07	4,00E-07	3,66E-07
25	2,00E-07	3,00E-07	2,75E-07	4,00E-07	6,00E-07	5,01E-07
32	3,00E-07	4,00E-07	3,80E-07	6,00E-07	8,00E-07	6,84E-07
40	4,00E-07	6,00E-07	5,13E-07	8,00E-07	1,00E-06	9,05E-07
50	6,00E-07	8,00E-07	7,07E-07	1,10E-06	1,40E-06	1,25E-06
63	9,00E-07	1,10E-06	9,93E-07	1,50E-06	1,80E-06	1,63E-06
79	1,30E-06	1,50E-06	1,42E-06	2,00E-06	2,40E-06	2,18E-06
100	1,90E-06	2,20E-06	2,07E-06	2,70E-06	3,10E-06	2,92E-06
126	2,80E-06	3,20E-06	3,03E-06	3,70E-06	4,10E-06	3,90E-06
158	4,10E-06	4,70E-06	4,43E-06	5,00E-06	5,50E-06	5,25E-06
200	6,30E-06	7,10E-06	6,66E-06	6,80E-06	7,40E-06	7,06E-06
251	9,40E-06	1,06E-05	9,94E-06	9,00E-06	9,80E-06	9,39E-06
316	1,43E-05	1,59E-05	1,51E-05	1,21E-05	1,29E-05	1,25E-05
398	2,19E-05	2,45E-05	2,30E-05	1,58E-05	1,71E-05	1,65E-05

501	3,38E-05	3,69E-05	3,53E-05	2,08E-05	2,22E-05	2,14E-05
631	5,25E-05	5,73E-05	5,46E-05	2,75E-05	3,80E-05	3,11E-05
794	8,13E-05	8,86E-05	8,46E-05	3,58E-05	4,38E-05	3,69E-05
1000	1,27E-04	1,37E-04	1,31E-04	4,58E-05	4,94E-05	4,77E-05
1259	1,99E-04	2,42E-04	2,07E-04	5,99E-05	6,62E-05	6,14E-05
1585	3,13E-04	3,78E-04	3,24E-04	7,88E-05	9,24E-05	8,08E-05
1995	4,92E-04	5,67E-04	5,09E-04	1,03E-04	1,16E-04	1,05E-04
2512	7,77E-04	8,62E-04	8,02E-04	1,34E-04	1,59E-04	1,38E-04
3162	1,23E-03	1,32E-03	1,26E-03	1,74E-04	2,00E-04	1,78E-04
3981	1,94E-03	2,08E-03	2,00E-03	2,29E-04	2,61E-04	2,34E-04
5012	3,08E-03	3,29E-03	3,16E-03	2,99E-04	3,43E-04	3,07E-04
6310	4,87E-03	5,11E-03	4,97E-03	3,89E-04	4,45E-04	4,06E-04
7943	7,77E-03	8,27E-03	7,95E-03	5,06E-04	5,78E-04	5,29E-04
10000	1,27E-02	3,48E-02	2,01E-02	6,54E-04	7,37E-04	6,76E-04
12589	2,68E-02	3,51E-02	2,78E-02	8,55E-04	9,71E-04	8,85E-04
15849	3,12E-02	4,40E-02	3,67E-02	1,11E-03	1,24E-03	1,15E-03
19953	4,93E-02	5,29E-02	5,02E-02	1,44E-03	1,59E-03	1,49E-03
25119	7,83E-02	1,09E-01	8,81E-02	1,88E-03	2,04E-03	1,94E-03
31623	1,24E-01	1,71E-01	1,34E-01	2,43E-03	2,64E-03	2,52E-03
39811	ND	ND	ND	3,17E-03	3,40E-03	3,26E-03
50119	ND	ND	ND	4,13E-03	4,46E-03	4,26E-03
63096	ND	ND	ND	5,34E-03	5,69E-03	5,48E-03
79433	ND	ND	ND	6,91E-03	7,34E-03	7,09E-03
100000	ND	ND	ND	8,94E-03	9,49E-03	9,16E-03
125893	ND	ND	ND	1,16E-02	1,23E-02	1,18E-02
158489	ND	ND	ND	1,50E-02	1,62E-02	1,54E-02
199526	ND	ND	ND	1,93E-02	2,16E-02	1,99E-02
251189	ND	ND	ND	2,47E-02	2,63E-02	2,53E-02
316228	ND	ND	ND	3,18E-02	3,37E-02	3,26E-02
398107	ND	ND	ND	4,11E-02	4,36E-02	4,21E-02
501187	ND	ND	ND	5,29E-02	5,60E-02	5,42E-02
630957	ND	ND	ND	6,83E-02	9,42E-02	7,07E-02
794328	ND	ND	ND	8,76E-02	1,00E-01	9,00E-02

Na aproximação de *insertion_sort* obteve-se um polinómio $1.234 * 10^{-10}n^2 + 3.342 * 10^{-7}n - 2.222 * 10^{-4}$. Sendo que para *Shell_sort* nem se fez aproximação a nenhum polinómio pois não se sabe qual o tipo de polinómio que melhor o aproxima.

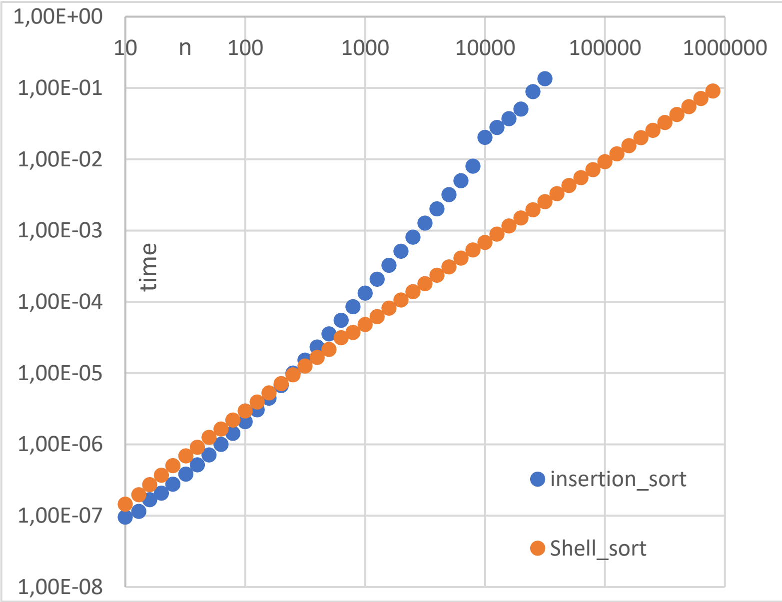

Gráfico 2

Pela análise do gráfico verificamos que o método *insertion_sort* é melhor que o método *Shell_sort* para valores até cerca dos 200 onde não se justifica a utilização do método mais complicado. No entanto, a partir daí verifica-se uma melhoria muito, mas muito significativa. Sendo que qualquer um destes métodos é bem melhor que os apresentados no ponto anterior.

O mais complicado para explicar será talvez o incremento súbito em *insertion_sort*, quando este alcançou os 10.000 o que poderá dever-se à diminuição do *boost* do processador devido ao aumento de temperatura ou então a algum processo do *Windows* que tenha consumido mais recursos.

Quicksort, Mergesort e Heapsort

Decidiram-se comparar estes métodos em conjunto pois apresentavam todos a mesma notação assintótica para o caso médio.

Tabela 4

	Quick sort			Merge sort			Heap sort		
n	min time	max time	avg time	min time	max time	avg time	min time	max time	avg time
10	0,00E+00	1,00E-07	9,63E-08	0,00E+00	1,00E-07	9,61E-08	1,00E-07	2,00E-07	1,50E-07
13	1,00E-07	2,00E-07	1,19E-07	1,00E-07	2,00E-07	1,20E-07	1,00E-07	3,00E-07	2,06E-07
16	1,00E-07	2,00E-07	1,68E-07	1,00E-07	2,00E-07	1,83E-07	2,00E-07	4,00E-07	2,87E-07
20	2,00E-07	3,00E-07	2,46E-07	2,00E-07	3,00E-07	2,12E-07	3,00E-07	5,00E-07	3,91E-07
25	3,00E-07	4,00E-07	3,33E-07	2,00E-07	3,00E-07	2,82E-07	5,00E-07	6,00E-07	5,40E-07
32	4,00E-07	5,00E-07	4,69E-07	3,00E-07	4,00E-07	3,88E-07	7,00E-07	8,00E-07	7,60E-07
40	6,00E-07	7,00E-07	6,31E-07	7,00E-07	9,00E-07	7,93E-07	9,00E-07	1,10E-06	1,01E-06
50	7,00E-07	9,00E-07	8,43E-07	9,00E-07	1,00E-06	9,76E-07	1,30E-06	1,50E-06	1,35E-06

63	1,00E-06	1,20E-06	1,13E-06	1,20E-06	1,30E-06	1,24E-06	1,70E-06	2,00E-06	1,83E-06
79	1,40E-06	1,70E-06	1,52E-06	1,70E-06	1,90E-06	1,84E-06	2,30E-06	2,60E-06	2,43E-06
100	1,90E-06	2,20E-06	2,04E-06	2,30E-06	2,70E-06	2,49E-06	3,10E-06	3,40E-06	3,23E-06
126	2,50E-06	2,90E-06	2,73E-06	3,00E-06	3,40E-06	3,16E-06	4,20E-06	4,50E-06	4,34E-06
158	3,40E-06	3,80E-06	3,59E-06	4,30E-06	4,70E-06	4,44E-06	5,50E-06	5,90E-06	5,71E-06
200	4,60E-06	5,00E-06	4,79E-06	5,70E-06	6,40E-06	5,97E-06	7,30E-06	8,00E-06	7,58E-06
251	6,00E-06	6,60E-06	6,32E-06	7,30E-06	7,80E-06	7,52E-06	9,70E-06	1,03E-05	9,99E-06
316	8,10E-06	9,00E-06	8,55E-06	9,90E-06	1,05E-05	1,02E-05	1,28E-05	1,34E-05	1,31E-05
398	1,08E-05	1,42E-05	1,13E-05	1,31E-05	1,38E-05	1,35E-05	1,70E-05	1,80E-05	1,76E-05
501	1,45E-05	1,59E-05	1,51E-05	1,65E-05	1,74E-05	1,69E-05	2,29E-05	2,40E-05	2,33E-05
631	1,93E-05	2,65E-05	2,04E-05	2,22E-05	2,31E-05	2,27E-05	2,91E-05	3,05E-05	2,96E-05
794	2,45E-05	3,33E-05	2,57E-05	2,99E-05	3,17E-05	3,06E-05	3,83E-05	4,92E-05	3,93E-05
1000	3,13E-05	3,36E-05	3,23E-05	3,74E-05	3,98E-05	3,86E-05	5,03E-05	5,36E-05	5,16E-05
1259	4,10E-05	4,34E-05	4,22E-05	5,01E-05	5,27E-05	5,15E-05	6,52E-05	7,55E-05	6,62E-05
1585	5,34E-05	5,64E-05	5,49E-05	6,63E-05	7,09E-05	6,71E-05	8,47E-05	9,06E-05	8,59E-05
1995	6,97E-05	7,37E-05	7,15E-05	8,44E-05	9,00E-05	8,63E-05	1,11E-04	1,18E-04	1,14E-04
2512	9,06E-05	9,52E-05	9,29E-05	1,10E-04	1,13E-04	1,11E-04	1,43E-04	1,64E-04	1,45E-04
3162	1,18E-04	1,42E-04	1,22E-04	1,48E-04	1,76E-04	1,53E-04	1,84E-04	2,09E-04	1,90E-04
3981	1,54E-04	1,74E-04	1,59E-04	1,87E-04	2,06E-04	1,91E-04	2,40E-04	2,87E-04	2,45E-04
5012	1,99E-04	2,29E-04	2,05E-04	2,42E-04	2,82E-04	2,47E-04	3,09E-04	3,46E-04	3,14E-04
6310	2,57E-04	3,05E-04	2,65E-04	3,25E-04	3,65E-04	3,32E-04	4,00E-04	4,47E-04	4,06E-04
7943	3,33E-04	3,81E-04	3,44E-04	4,11E-04	4,61E-04	4,18E-04	5,17E-04	5,72E-04	5,24E-04
10000	4,31E-04	4,88E-04	4,46E-04	5,26E-04	5,92E-04	5,35E-04	6,67E-04	7,35E-04	6,79E-04
12589	5,57E-04	6,20E-04	5,76E-04	7,09E-04	7,70E-04	7,19E-04	8,63E-04	9,38E-04	8,79E-04
15849	7,18E-04	8,04E-04	7,41E-04	8,96E-04	9,79E-04	9,14E-04	1,12E-03	1,20E-03	1,14E-03

19953	9,26E-04	1,02E-03	9,55E-04	1,14E-03	1,23E-03	1,16E-03	1,44E-03	1,54E-03	1,48E-03
25119	1,20E-03	1,30E-03	1,24E-03	1,53E-03	1,64E-03	1,56E-03	1,86E-03	2,04E-03	1,90E-03
31623	1,54E-03	1,68E-03	1,59E-03	1,94E-03	2,10E-03	2,00E-03	2,40E-03	2,54E-03	2,44E-03
39811	1,98E-03	2,14E-03	2,04E-03	2,46E-03	2,63E-03	2,51E-03	3,09E-03	3,30E-03	3,16E-03
50119	2,54E-03	2,71E-03	2,60E-03	3,30E-03	3,52E-03	3,37E-03	3,98E-03	4,21E-03	4,06E-03
63096	3,25E-03	3,48E-03	3,35E-03	4,17E-03	4,44E-03	4,26E-03	5,15E-03	5,49E-03	5,27E-03
79433	4,17E-03	4,49E-03	4,30E-03	5,31E-03	5,68E-03	5,41E-03	6,64E-03	6,99E-03	6,74E-03
100000	5,31E-03	5,62E-03	5,44E-03	7,07E-03	7,50E-03	7,21E-03	8,60E-03	9,09E-03	8,74E-03
125893	6,75E-03	7,23E-03	6,94E-03	9,16E-03	9,79E-03	9,38E-03	1,12E-02	1,18E-02	1,14E-02
158489	8,54E-03	9,05E-03	8,76E-03	1,15E-02	1,20E-02	1,17E-02	1,45E-02	1,53E-02	1,47E-02
199526	1,08E-02	1,14E-02	1,10E-02	1,56E-02	1,65E-02	1,59E-02	1,88E-02	1,98E-02	1,91E-02
251189	1,35E-02	1,44E-02	1,39E-02	1,96E-02	2,05E-02	1,99E-02	2,44E-02	2,56E-02	2,48E-02
316228	1,70E-02	1,80E-02	1,74E-02	2,42E-02	2,56E-02	2,46E-02	3,16E-02	3,32E-02	3,20E-02
398107	2,13E-02	2,28E-02	2,19E-02	3,26E-02	3,45E-02	3,32E-02	4,09E-02	4,29E-02	4,14E-02
501187	2,67E-02	2,83E-02	2,73E-02	4,09E-02	4,29E-02	4,14E-02	5,29E-02	5,51E-02	5,35E-02
630957	3,34E-02	3,55E-02	3,42E-02	5,05E-02	5,26E-02	5,11E-02	6,81E-02	7,11E-02	6,90E-02
794328	4,20E-02	4,45E-02	4,30E-02	6,79E-02	7,08E-02	6,88E-02	8,78E-02	9,18E-02	8,90E-02
1000000	5,26E-02	5,54E-02	5,37E-02	8,50E-02	8,82E-02	8,59E-02	1,14E-01	1,20E-01	1,15E-01
1258925	6,61E-02	6,97E-02	6,75E-02	1,05E-01	1,09E-01	1,06E-01	ND	ND	ND
1584893	8,30E-02	8,77E-02	8,48E-02	ND	ND	ND	ND	ND	ND
1995262	1,04E-01	1,10E-01	1,07E-01	ND	ND	ND	ND	ND	ND

No caso de *quick_sort* verificámos que um bom polinómio é $9.833 * 10^{-9} * n * \log(0.2048 * n) + 1.629 * 10^{-4}$, para *merge_sort* obteve-se $1.106 * 10^{-8} * n * \log(48.38 * n) - 1.073 * 10^{-4}$ e finalmente para *heap_sort* teve-se $2.073 * 10^{-8} * n * \log(0.3135 * n) - 6.056 * 10^{-5}$.

Nesta situação a distinção já é mais difícil de se realizar visualmente sendo mais simples analisar os polinómios obtidos. No entanto, todos os métodos apresentam uma velocidade de desempenho relativamente próxima.

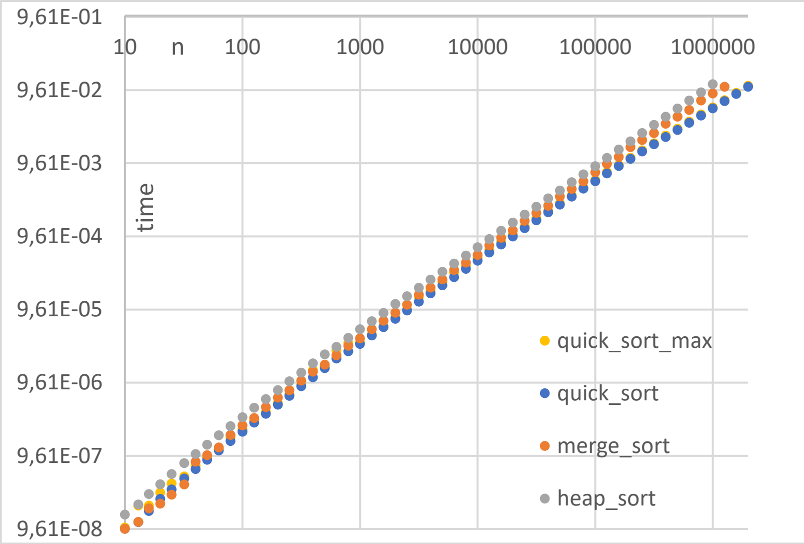

Gráfico 3

Para números baixos reparamos que *merge_sort* é melhor até cerca de 40 elementos, depois deste valor o método *quick_sort* é melhor. Do pior dos 3 métodos *heap_sort* é o que apresenta sempre piores resultados. Sendo que nem apresenta maior facilidade de implementação. No entanto, é de referir o uso de memória extra que *merge_sort* requer pelo que *heap_sort* poderá até ser um método preferível devido a não fazer tal uso. Um pormenor que também é de referir é que o pior caso para *quick_sort* apresenta ordem n^2. Aspeto que não se conseguiu verificar neste gráfico mesmo comparando o tempo máximo de *quick_sort* com os tempos médios dos outros.

Rank_sort e Selection_sort

Como estas rotinas apresentam ordem fixa de n^2 é curioso verificar-se como se comparam uma à outra. E de que forma o uso de memória extra requerido por *rank_sort* é vantajoso.

Tabela 5

	Rank sort			Selection sort		
n	min time	max time	avg time	min time	max time	avg time
10	2,00E-07	3,00E-07	2,19E-07	1,00E-07	2,00E-07	1,55E-07
13	2,00E-07	3,00E-07	2,19E-07	2,00E-07	3,00E-07	2,43E-07
16	3,00E-07	6,00E-07	4,96E-07	3,00E-07	4,00E-07	3,88E-07
20	4,00E-07	6,00E-07	5,69E-07	5,00E-07	6,00E-07	5,88E-07
25	5,00E-07	8,00E-07	7,28E-07	9,00E-07	1,00E-06	9,01E-07
32	7,00E-07	1,00E-06	8,95E-07	1,40E-06	1,50E-06	1,47E-06
40	9,00E-07	1,10E-06	9,79E-07	2,20E-06	2,30E-06	2,26E-06
50	1,50E-06	1,70E-06	1,54E-06	3,40E-06	3,60E-06	3,49E-06
63	2,20E-06	2,40E-06	2,30E-06	5,40E-06	5,60E-06	5,51E-06
79	3,30E-06	3,70E-06	3,51E-06	8,40E-06	8,80E-06	8,60E-06
100	5,30E-06	6,40E-06	5,59E-06	1,34E-05	1,39E-05	1,37E-05
126	8,30E-06	9,10E-06	8,64E-06	2,13E-05	2,21E-05	2,17E-05
158	1,29E-05	1,41E-05	1,34E-05	3,45E-05	3,66E-05	3,52E-05
200	2,00E-05	2,16E-05	2,08E-05	5,40E-05	5,87E-05	5,47E-05
251	3,15E-05	3,38E-05	3,26E-05	8,52E-05	9,06E-05	8,67E-05
316	4,99E-05	5,43E-05	5,16E-05	1,36E-04	1,56E-04	1,38E-04

398	7,93E-05	8,40E-05	8,14E-05	2,16E-04	2,37E-04	2,18E-04
501	1,26E-04	1,33E-04	1,29E-04	3,43E-04	3,76E-04	3,47E-04
631	2,00E-04	2,25E-04	2,07E-04	5,45E-04	5,91E-04	5,50E-04
794	3,17E-04	3,59E-04	3,27E-04	8,66E-04	9,33E-04	8,78E-04
1000	5,03E-04	5,69E-04	5,17E-04	1,38E-03	1,47E-03	1,40E-03
1259	7,98E-04	8,88E-04	8,21E-04	2,19E-03	2,31E-03	2,22E-03
1585	1,27E-03	1,37E-03	1,30E-03	3,47E-03	3,83E-03	3,56E-03
1995	2,01E-03	2,15E-03	2,06E-03	5,51E-03	5,83E-03	5,61E-03
2512	3,20E-03	3,71E-03	3,31E-03	8,76E-03	9,13E-03	8,87E-03
3162	5,08E-03	5,56E-03	5,24E-03	1,39E-02	1,47E-02	1,41E-02
3981	8,07E-03	8,48E-03	8,23E-03	2,21E-02	2,30E-02	2,24E-02
5012	1,29E-02	1,34E-02	1,31E-02	3,51E-02	3,66E-02	3,55E-02
6310	2,04E-02	2,16E-02	2,08E-02	5,57E-02	5,81E-02	5,64E-02
7943	3,24E-02	3,38E-02	3,29E-02	8,84E-02	9,18E-02	8,94E-02
10000	5,15E-02	5,37E-02	5,22E-02	1,40E-01	1,46E-01	1,42E-01
12589	8,17E-02	8,52E-02	8,28E-02	ND	ND	ND
15849	1,30E-01	1,35E-01	1,31E-01	ND	ND	ND

Para *rank_sort* obtivemos um polinómio excelente que apresentou um r^2 de 1 e tivemos como polinómio $5.215 * 10^{-10}n^2 + 4.683 * 10^{-9}n - 3.795 * 10^{-6}$. E para *selection_sort* também se alcançou um r^2 de 1 com o polinómio $1.425 * 10^{-9}n^2 - 5.144 * 10^{-8}n + 1.046 * 10^{-5}$.

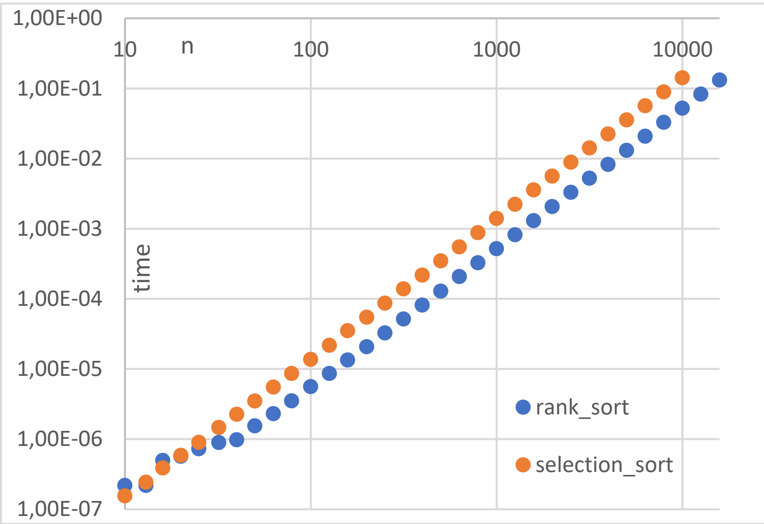

Gráfico 4

Nesta situação verifica-se que *rank_sort* ao início oscila bastante ao contrário do outro método, sendo que o seu desempenho se torna difícil de medir e comparar, no entanto quando se passa de n igual a 32 verifica-se que é um método mais rápido e a partir de n igual a 40 não ocorrem tais desvios. É de mencionar que este método requer o uso de memória extra o que o pode tornar indesejável consoante a aplicação.

Tree_sort

Tabela 6

	Tree sort		
n	min time	max time	avg time
10	4,00E-07	6,00E-07	5,01E-07
13	6,00E-07	8,00E-07	6,58E-07
16	7,00E-07	1,00E-06	8,41E-07
20	1,00E-06	1,30E-06	1,09E-06
25	1,30E-06	1,60E-06	1,40E-06
32	1,70E-06	2,10E-06	1,87E-06
40	2,20E-06	2,70E-06	2,41E-06
50	2,90E-06	3,40E-06	3,11E-06
63	3,80E-06	4,50E-06	4,04E-06
79	4,90E-06	8,20E-06	5,21E-06
100	6,40E-06	3,45E-05	6,88E-06
126	8,30E-06	3,68E-05	9,33E-06
158	1,07E-05	3,96E-05	1,22E-05
200	1,39E-05	4,49E-05	1,61E-05
251	1,78E-05	4,87E-05	2,12E-05
316	2,30E-05	5,63E-05	2,76E-05
398	2,98E-05	6,30E-05	3,59E-05
501	3,83E-05	7,37E-05	4,66E-05
631	4,94E-05	8,75E-05	6,07E-05
794	6,39E-05	1,03E-04	7,76E-05
1000	8,30E-05	1,25E-04	1,00E-04
1259	1,07E-04	1,54E-04	1,30E-04
1585	1,40E-04	1,88E-04	1,67E-04
1995	1,87E-04	2,46E-04	2,18E-04
2512	2,64E-04	3,30E-04	2,81E-04
3162	3,35E-04	4,32E-04	3,66E-04
3981	4,37E-04	5,30E-04	4,75E-04
5012	5,77E-04	6,93E-04	6,15E-04
6310	7,55E-04	8,75E-04	7,99E-04
7943	9,86E-04	1,14E-03	1,05E-03
10000	1,32E-03	1,55E-03	1,39E-03
12589	1,75E-03	1,98E-03	1,83E-03
15849	2,31E-03	2,60E-03	2,43E-03
19953	3,07E-03	3,42E-03	3,21E-03
25119	4,07E-03	4,99E-03	4,26E-03
31623	5,38E-03	6,03E-03	5,59E-03
39811	7,15E-03	8,21E-03	7,46E-03
50119	9,60E-03	1,29E-02	1,02E-02
63096	1,28E-02	2,07E-02	1,45E-02
79433	1,69E-02	2,99E-02	1,92E-02

100000	2,26E-02	4,28E-02	2,62E-02
125893	3,07E-02	5,48E-02	3,53E-02
158489	4,30E-02	6,55E-02	4,92E-02
199526	6,27E-02	8,78E-02	7,05E-02
251189	9,71E-02	1,62E-01	1,10E-01

Neste caso esperávamos um bom polinómio de ajuste como foi anteriormente dito de ordem $n \ log \ n + n$ mas verificámos que esta função não se enquadrou bem, um r^2 bastante mau, e fazendo um ajuste a um polinómio quadrático obtiveram-se melhores valores, pelo que concluímos que a inserção não ocorre em média em ordem $log \ n$ e sim em ordem n pelo que se fez a aproximação a um polinómio quadrado. Obtendo-se então $1.179 * 10^{-12}n^2 + 1.33 * 10^{-7}n - 3.956 * 10^{-5}$.

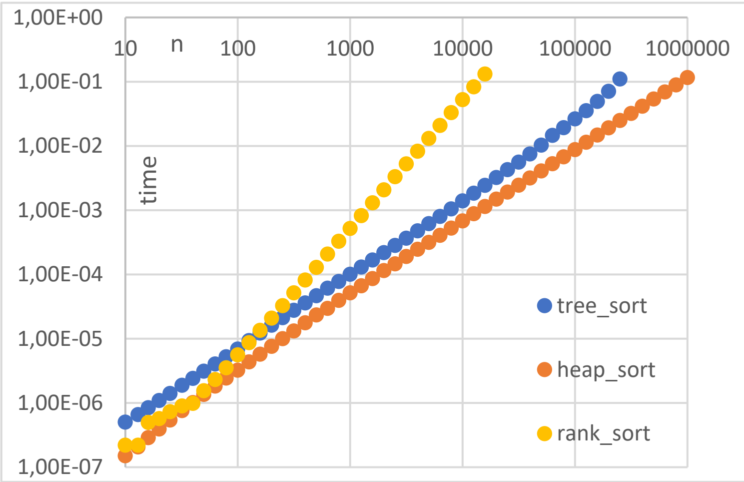

Gráfico 5

Comparando com um método quadrático, *rank_sort*, verifica-se que *tree_sort* é melhor, mas que não chega a ser tão bom como os métodos $n \ log \ n$.

Bogo_sort

Nesta situação devida à complexidade da função tivemos que fazer alterações à forma de como se faziam os testes, passando a fazer-se o estudo de 1 a 11 elementos com o incremento de 1, ao contrário dos outros métodos.

Tabela 7

	Bogo sort		
n	min time	max time	avg time
1	0,00E+00	1,00E-07	9,40E-09
2	0,00E+00	2,00E-07	6,20E-08
3	0,00E+00	9,00E-07	2,76E-07
4	1,00E-07	5,60E-06	1,60E-06
5	5,00E-07	3,82E-05	1,09E-05
6	3,40E-06	2,44E-04	7,17E-05
7	2,61E-05	2,12E-03	5,89E-04
8	3,11E-04	2,56E-02	6,61E-03
9	3,96E-03	3,00E-01	7,92E-02
10	3,93E-02	3,15E+00	9,09E-01
11	4,52E-01	2,69E+01	8,11E+00

Tendo em conta a natureza da função não faz sentido encontrar polinómio que se aproxime, pois esperamos $n!$ e uma função exponencial não lhe faz justiça.

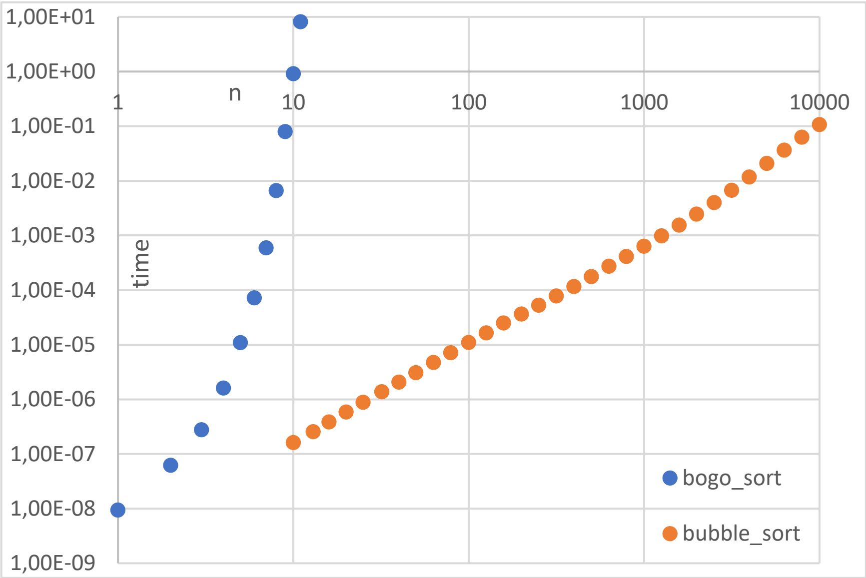

Gráfico 6

Passando a analisar o gráfico verificamos que nem o pior dos métodos apresentados anteriormente, *bubble_sort*, é tão mau quanto *bogo_sort* por isso é mesmo um método que para além de ter uma implementação mais complexa ainda apresenta um tempo de execução impraticável para qualquer uso. Posto isso para facilidade de implementação e bom desempenho *insertion_sort* é melhor que ambos estes métodos.

Geral

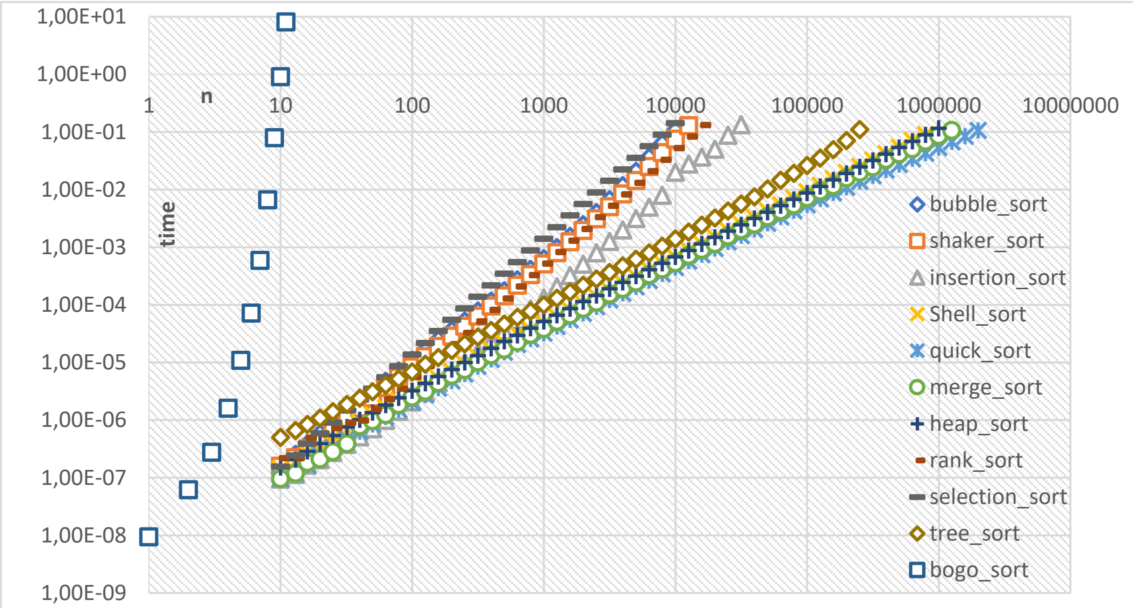

Gráfico 7

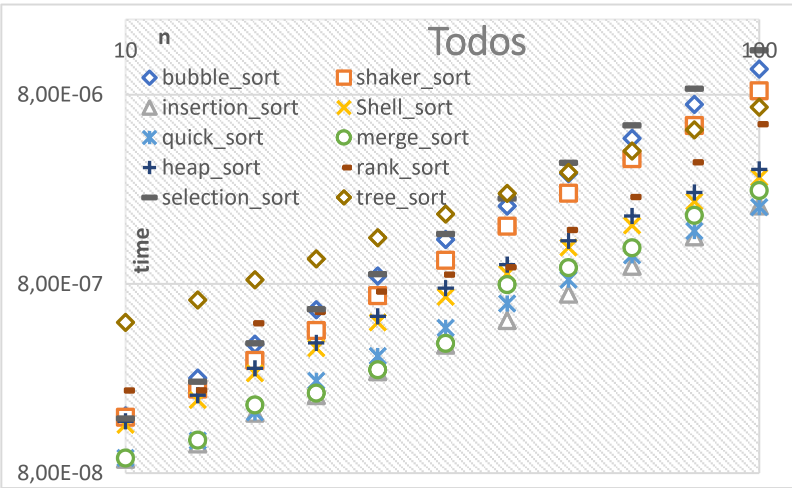

Gráfico 8

Comparando todos os métodos é possível verificar-se que *bogo_sort* é de facto impraticável. E que para os números maiores *quick_sort* acaba por ser a que demora menos tempo, pelo que poderá ser de facto a melhor rotina para números muito elevados.

Para valores de n entre os 10 e os 100 verificamos que *tree_sort* apresenta-se como sendo o pior e para valores reduzidos *merge_sort*, *quick_sort* e *insertion_sort* apresentam valores exatamente iguais, o que se esperava uma vez que estes dois primeiros métodos fazem recurso a *insertion_sort* para valores baixos, 40 e 20, respetivamente. É possível analisar-se o crescimento mais elevado de *rank_sort* em comparação a *heap_sort* para n superior a 50 e a proximidade que passa a ter com *Shell_sort*.

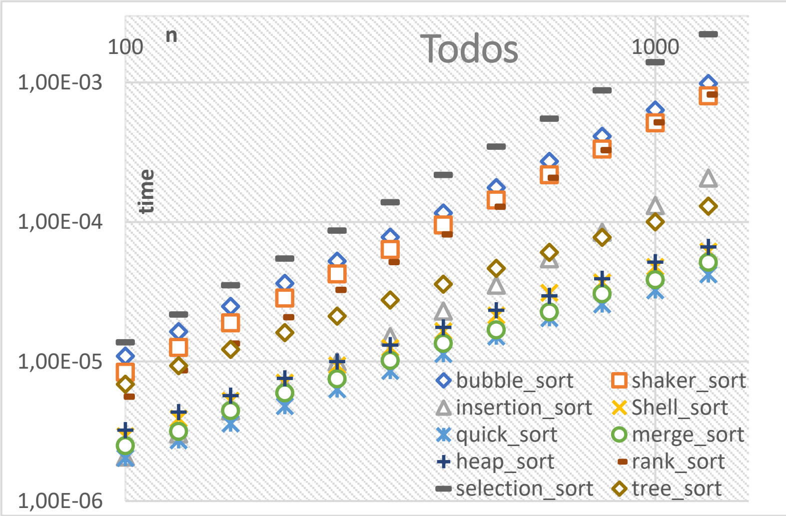

Gráfico 9

Para valores de n entre 100 e 1000 verifica-se uma maior separação em que se pode verificar o crescimento quadrático para *selection_sort*, *bubble_sort*, *shaker_sort* e *rank_sort*, que se tornam mais lentos. Verificando-se que *tree_sort* em n igual a 794 tem um tempo igual a *insertion_sort* passando a ter um desempenho superior a partir daí. É de mencionar que se verifica o desempenho inferior que *selection_sort* desenvolve e a separação que os métodos $n \, log \, n$ passam a ter dos métodos quadráticos. Para valores de n superiores não se apresentam diferenças substanciais que sejam de referir.

Conclusão

Com a realização deste trabalho é possível fazer-se a escolha do algoritmo de ordenação a usar consoante o número de elementos esperado. Para valores menores que 100 provavelmente iria-se escolher *insertion_sort* devido à facilidade de implementação, bom desempenho e não fazer uso de memória extra. Para valores de n superiores a 100 *quick-sort* apresenta-se como sendo bastante bom. No entanto, é preciso ter atenção que o pior caso será ordem n^2, mas também era preciso algum azar em escolher-se sempre o pior *pivot*, pelo que uma aposta mais segura poderá ser *heap_sort* isto por não fazer uso de memória extra. Mas também *Shell_sort* apresenta um muito bom desempenho, podendo vir a perder para números maiores, mas tal não se conseguiu verificar com os nossos estudos sendo necessário fazer testes para valores de n maiores. Afirmamos isto por se terem analisado alguns estudos que tentaram obter um polinómio deste método, de forma $n^{\alpha}, 1 < \alpha < 2$, pelo que $n \, log \, n$ é melhor.

Consideramos que este trabalho foi um bom momento de estudo para uma melhor programação futura de forma a se conseguir tomar boas decisões consoante o método a aplicar, tendo em conta os benefícios e desvantagens de cada um dos métodos e dos recursos disponíveis, devido à relevância do uso de microprocessadores atualmente e devido à pouca memória disponível é então muito relevante.

Bibliografia

Material de aula disponibilizado no eLearning

https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf

https://software.intel.com/content/www/us/en/develop/articles/an-efficient-parallel-three-way-quicksort-using-intel-c-compiler-and-openmp-45-library.html

Código

*Tree*_sort

```
#include "sorting_methods.h"

#include <stdlib.h>

struct tree_node

{

   struct tree_node *left;

   T data;

   struct tree_node *right;

};

void insert(struct tree_node **tr, T data)

{

   if( *tr == NULL)

   {

      (*tr)=malloc(sizeof(struct tree_node));

      (*tr)->left = NULL;

      (*tr)->right = NULL;

      (*tr)->data = data;

   }

   else

   {

      if(data < (*tr)->data)

         insert( &((*tr)->left),data);

      else

         insert( &((*tr)->right),data);

   }

}

int counter_insert;


void order(int *data,struct tree_node *link)

{

   if(link != NULL)

   {

      order(data,link->left);

      data[counter_insert++] = link->data;

      order(data,link->right);

   }

}
```

```
void tree_sort(T *data, int first, int one_after_last)
{
    struct tree_node *root;
    root = NULL;
    for(int i=first; i<one_after_last;i++)
    {
        insert(&root,data[i]);
    }
    counter_insert = first;
    order(data,root);
}
```

Bogo_sort

```
#include "sorting_methods.h"
#include <stdlib.h>
#include <stdio.h>

int sorted(T *data, int first, int one_after_last)
{
    for(int i = first; i < one_after_last-1;i++)
    {
        if(data[i] > data[i+1])
            return -1;
    }
    return 1;
}

void shuffle(T *data, int first, int one_after_last)
{
    for(int i=first; i<one_after_last; i++)
    {
        int idx = rand()%(one_after_last-first) + first;
        int tmp = data[i];
        data[i] = data[idx];
        data[idx] = tmp;
    }
}
void bogo_sort(T *data, int first, int one_after_last)
{
```

```
    while(sorted(data,first,one_after_last) == -1)
    {
        shuffle(data,first,one_after_last);
    }
}
```