RELATÓRIO

Sobre a eficiência de diferentes algoritmos de ordenação

INTRODUÇÃO

O objetivo desse relatório é comparar a eficiência de quatro diferentes tipos de algoritmos de ordenação, dois deles de complexidade n^2 (chamados “Bubble Sort” e “Insertion Sort”) e dois deles de complexidade *nlogn* (“Heap Sort” e “Merge Sort”).

Foi desenvolvido um programa em Java – de nome “Sorting Efficiency” – que cria vários *arrays* automaticamente (de diferentes tamanhos, graus de dificuldade de ordenação, e tipos de dados) e, após isso, todos os algoritmos de ordenação citados no parágrafo anterior os ordenam e calculam os tempos necessários para fazê-lo. Serão analisados, aqui, os resultados obtidos após a execução deste programa e serão feitas considerações acerca deles. É importante ressaltar que, neste relatório, **todos os gráficos e tabelas possuirão tempos em segundos.**

VISÃO GERAL DO SOFTWARE

Primeiramente, é interessante contextualizar o leitor acerca das capacidades do software desenvolvido em Java sem entrar em detalhes muito técnicos (pois sairia do objetivo principal deste relatório).

O software permite que o usuário escolha os tamanhos dos *arrays* que serão ordenados e também o número de tipos de dados que eles guardarão. Para esse relatório, os tamanhos dos *arrays* criados inicialmente serão múltiplos de dez no intervalo de dez até um milhão (inclusive) e os tipos de dados escolhidos serão três (tipos *int*, *float* e *double*). Após isso, para cada tamanho e tipo de dado, são criados três *arrays* de dificuldades diferentes: o primeiro possui seus números gerados de forma completamente randômica, o segundo possui a ordem invertida em relação a ordem que queremos ordenar e o terceiro está quase ordenado, porém com alguns números fora de sua ordem original.

Gerados esses *arrays*, os algoritmos de ordenação irão, um a um, ordená-los e mostrarão o tempo gasto para fazê-lo no console além de gerarem arquivos .csv com os resultados para que possam ser utilizados para criação de gráficos (funciona muito bem no Google Planilhas, por exemplo).

RESULTADOS: MERGE SORT

Dos resultados do *Merge Sort*, percebemos que ele é um algoritmo bastante eficaz. É capaz de ordenar um milhão de elementos em um tempo razoavelmente curto. Uma característica perceptível desse algoritmo é o fato de que ele é muito eficaz quando a ordem do *array* a ser ordenado é invertida, de modo que o tempo calculado, nesse caso, muitas vezes é menor do que o tempo referente à ordenação do caso quase ordenado. Isso é bem visível no gráfico da ordenação de um milhão de termos no final dessa seção.

Tempos de Ordenação: Números Inteiros

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.000150699	0.0000853	0.0000537
100	0.000579799	0.000476899	0.0004126
1000	0.0056191	0.003845299	0.0042355
10000	0.0299638	0.0088639	0.025553199
100000	0.1762831	0.1178873	0.155692
1000000	2.457160501	1.6962321	2.015312199

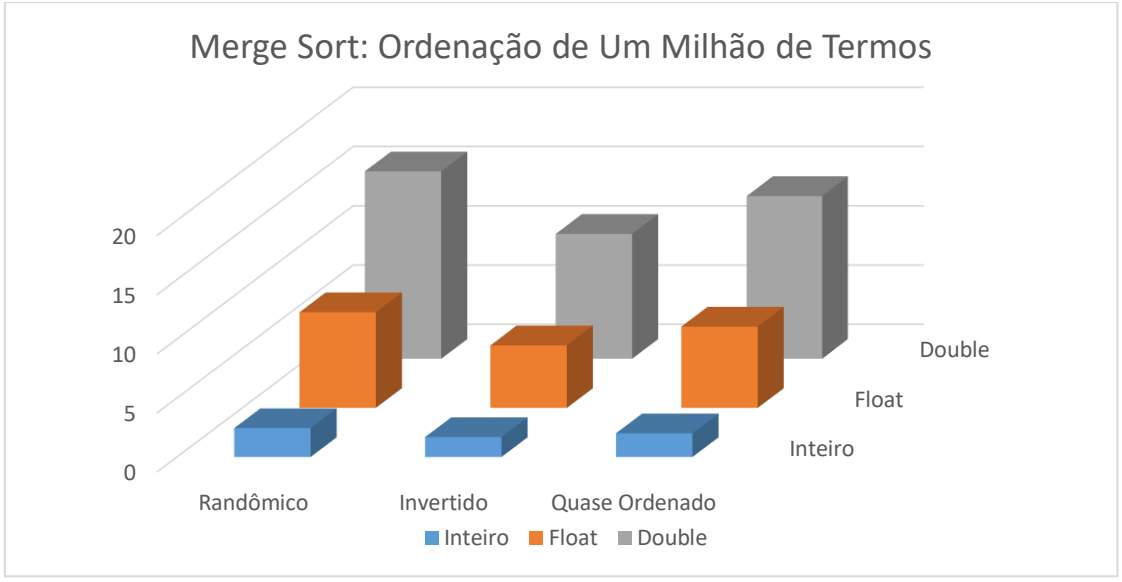
Tempos de Ordenação: Números Float

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0004343	0.000324601	0.0001978
100	0.0013242	0.0010095	0.000938999
1000	0.0100833	0.010865101	0.0090386
10000	0.0547742	0.0393481	0.0426022
100000	0.6386841	0.428060599	0.540526999
1000000	8.096122101	5.300737801	6.8794686

Tempos de Ordenação: Números Double

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.000413599	0.000252499	0.000161701
100	0.0022678	0.001602101	0.001713799
1000	0.0108751	0.0085072	0.010759801
10000	0.0995411	0.079822399	0.0840645
100000	1.253752	0.8484362	1.080085401
1000000	15.8147043	10.544693	13.7395942

Gráfico



RESULTADOS: HEAP SORT

Em relação ao *Heap Sort*, podemos dizer que ele é bem eficiente também, mas não tanto quanto o *Merge Sort*. Isso é visível quando se analisa o último gráfico desta seção (nesse gráfico, somente foram comparados os casos em que o *array* a ser ordenado possuía tamanho maior que mil pois, para tamanhos menores, os tempos de execução são muito parecidos para ambos os algoritmos) e percebe-se que, quando o *array* é maior que dez mil, há uma diferença de tempo notável entre o *Merge Sort* e o *Heap Sort*.

É notável também, analisando o primeiro gráfico desta seção, que o *Heap Sort*, assim como o *Merge Sort*, parece resolver a ordenação de *arrays* invertidos de forma mais rápida, mas percebe-se que essa característica é mais evidente no *Merge Sort*.

Tempos de Ordenação: Números Inteiros

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0002792	0.0001215	0.0001583
100	0.0006914	0.000490001	0.000637801
1000	0.006718301	0.006604201	0.007295001
10000	0.0399172	0.0164511	0.0173536
100000	0.2673832	0.2255011	0.238509199
1000000	3.9367158	3.033367899	3.181123

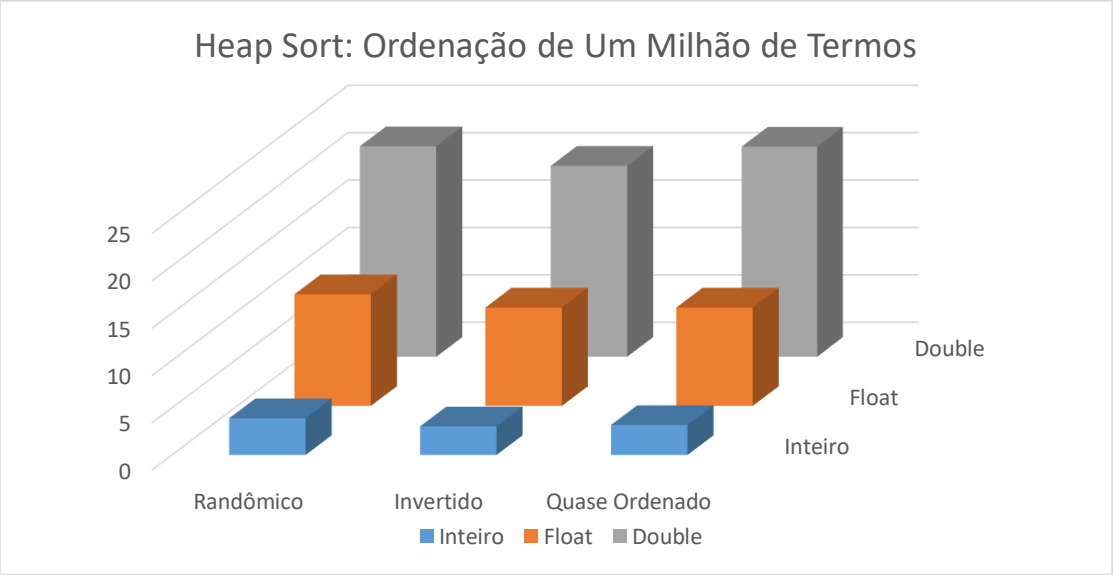
Tempos de Ordenação: Números Float

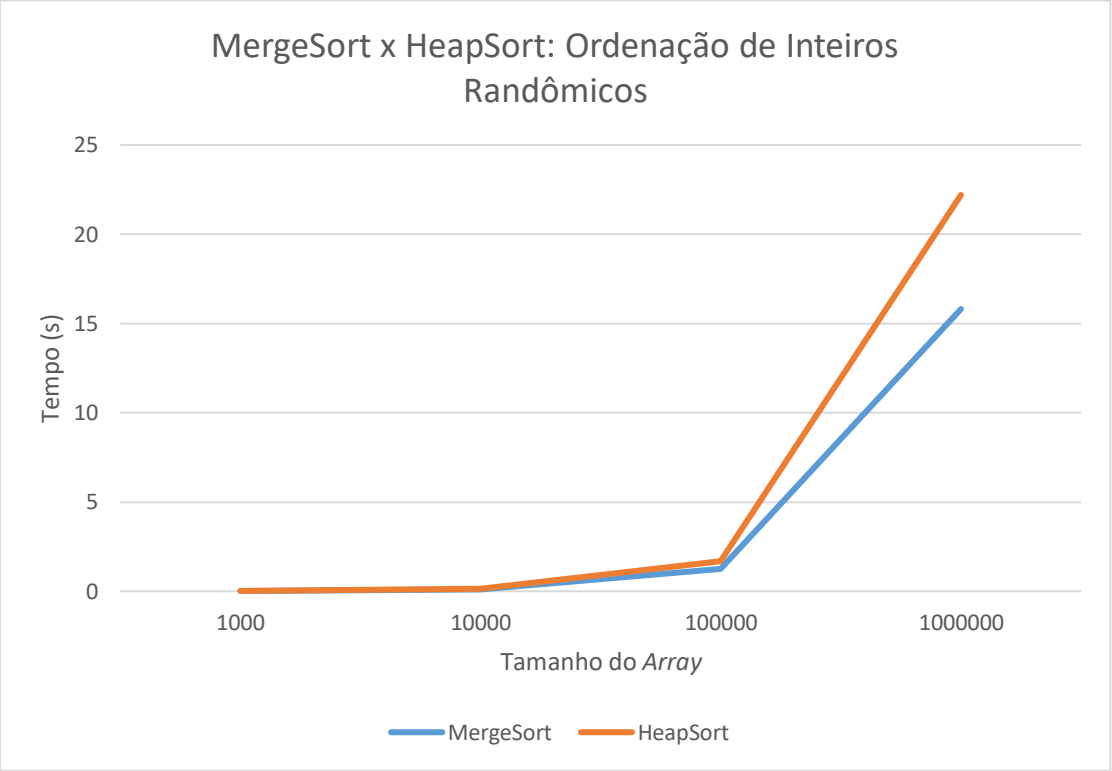
Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.002052299	0.000418	0.000411301
100	0.002032099	0.001398599	0.0013306
1000	0.018406299	0.0126997	0.015316799
10000	0.0784547	0.066235001	0.0836784
100000	0.8849931	0.8160997	0.8263614
1000000	11.8139096	10.3926918	10.445358901

Tempos de Ordenação: Números Double

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0005227	0.000385801	0.000253401
100	0.003151	0.002478499	0.0037206
1000	0.0171741	0.013735199	0.0177225
10000	0.1327448	0.133576301	0.150787401
100000	1.694155599	1.590173899	1.667381
1000000	22.217160101	20.1464596	22.1647984

Gráficos





RESULTADOS: INSERTION SORT

Agora fica clara a melhor eficiência dos algoritmos de complexidade *nlogn* acima. O *Insertion Sort*, por ser de complexidade n^2 é muito mais lento do que os dois algoritmos já discutidos. Essa má eficiência torna-se evidente em casos com muitos elementos: O *Merge Sort* e o *Heap Sort* ordenaram um milhão de elementos mais rápido do que o *Insertion Sort* ordenou cem mil deles. Além disso, é importante ressaltar que este algoritmo não conseguiu, com as capacidades computacionais disponíveis ao autor deste relatório, ordenar um milhão de elementos em tempo razoável (e por isso, não temos este tamanho mostrado nas tabelas).

Uma outra característica notável é que este algoritmo se destaca brilhantemente na parte de ordenar *arrays* quase ordenados e sofre quando precisa lidar com *arrays* em ordem invertida. Isso é perceptível visualmente no gráfico final desta seção.

Tempos de Ordenação: Números Inteiros

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0000826	0.0001311	0.000044
100	0.0169072	0.0015474	0.000113
1000	0.0268402	0.0740503	0.0029632
10000	1.4859568	3.3238385	0.1966762
100000	182.96437	345.7307431	23.8242903

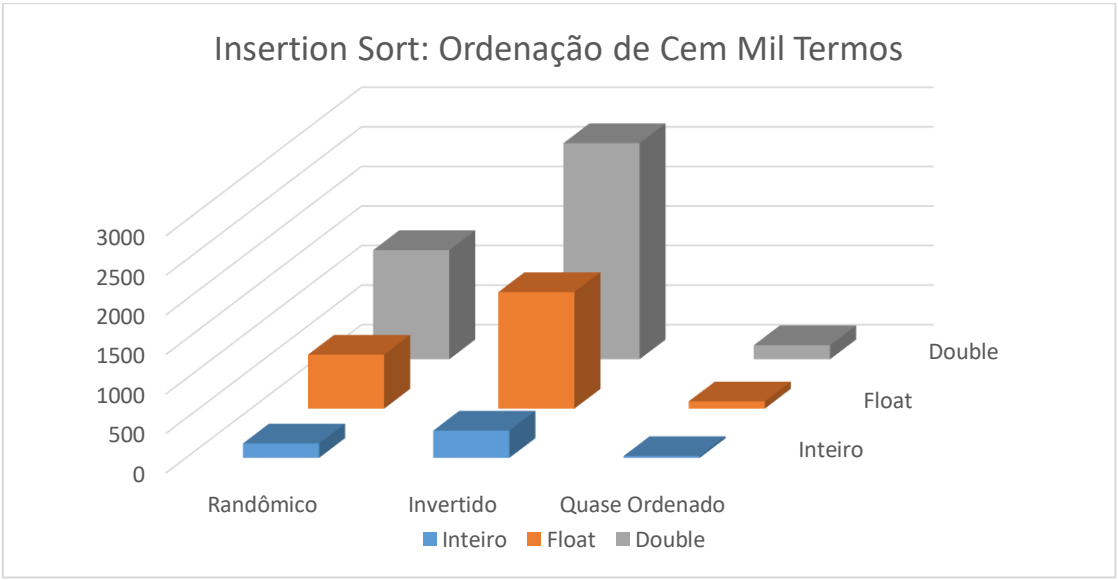
Tempos de Ordenação: Números Float

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.000367	0.0005622	0.0001118
100	0.0022323	0.0064021	0.0003818
1000	0.0632421	0.123179	0.0084071
10000	6.7292597	13.6216178	0.8742792
100000	682.8920625	1473.2131793	90.5826941

Tempos de Ordenação: Números Double

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0002839	0.0003914	0.0000823
100	0.0033894	0.0035528	0.0003801
1000	0.1308942	0.2613898	0.0180773
10000	13.3371467	26.6183176	1.7748459
100000	1378.881024	2730.0857843	177.4872953

Gráfico



RESULTADOS: BUBBLE SORT

Por fim, tem-se o último e menos eficiente algoritmo de ordenação deste relatório, o *Bubble Sort*. É perceptível, analisando os dados, que o *Bubble Sort*, quando ordena *arrays* invertidos, apresenta resultados muito parecidos aos do *Insertion Sort*. Isso faz certo sentido pois seria a situação de maior dificuldade. O problema é que o *Bubble Sort* parece não ser capaz de melhorar seu tempo em casos em que deveria fazer menos comparações (como, por exemplo, no caso quase ordenado), o que mostra que este algoritmo é bem simples e primário.

No primeiro gráfico do fim dessa seção, pode-se ver como o *Bubble Sort* possui desempenhos semelhantes não importando a dificuldade do *array* a ser ordenado. No segundo gráfico, tem-se uma comparação entre ele e o *Insertion Sort* e podemos ver o quanto este último é mais eficiente em casos maiores.

É importante ressaltar também que, assim como o *Insertion Sort*, o *Bubble Sort* foi incapaz de completar uma ordenação de um milhão de elementos em tempo razoável e, por isso, não há dados para esse caso nos números abaixo.

Tempos de Ordenação: Números Inteiros

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.000304	0.0001547	0.0000849
100	0.0022142	0.0019494	0.0008671
1000	0.0642145	0.0470135	0.0274012
10000	3.3629003	3.1399449	2.8993898
100000	398.9152238	382.683484	347.7259777

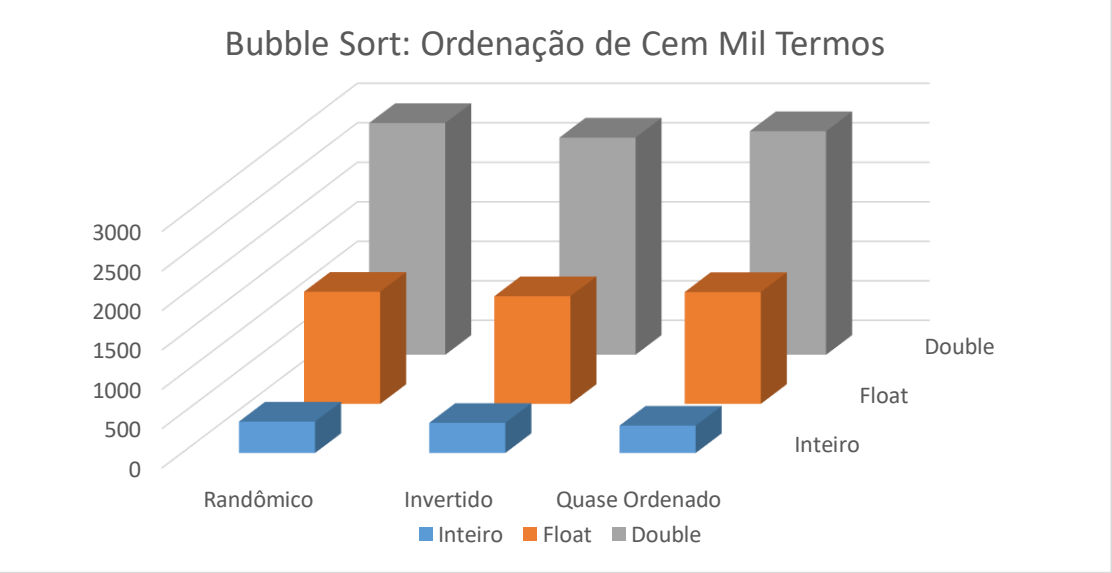
Tempos de Ordenação: Números Float

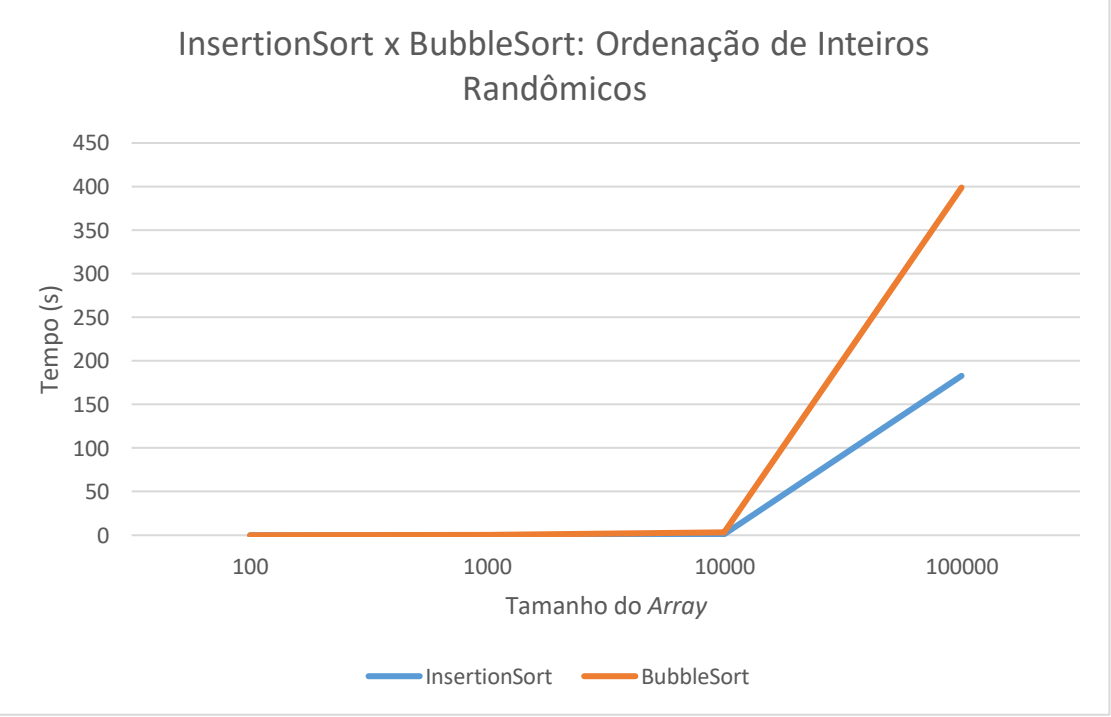
Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.001253	0.0005658	0.0003797
100	0.0050481	0.0077308	0.0033445
1000	0.1500137	0.1331604	0.1434943
10000	14.3418745	13.2765351	13.6742102
100000	1422.3218611	1365.6265906	1418.1506149

Tempos de Ordenação: Números Double

Tamanho	*Randômico*	*Invertido*	*Quase Ordenado*
10	0.0005261	0.0004402	0.000312
100	0.0086938	0.0047193	0.0031284
1000	0.2875252	0.2648674	0.2838534
10000	28.0911914	27.0376442	27.7862813
100000	2932.6641787	2748.3518052	2829.267995

Gráficos

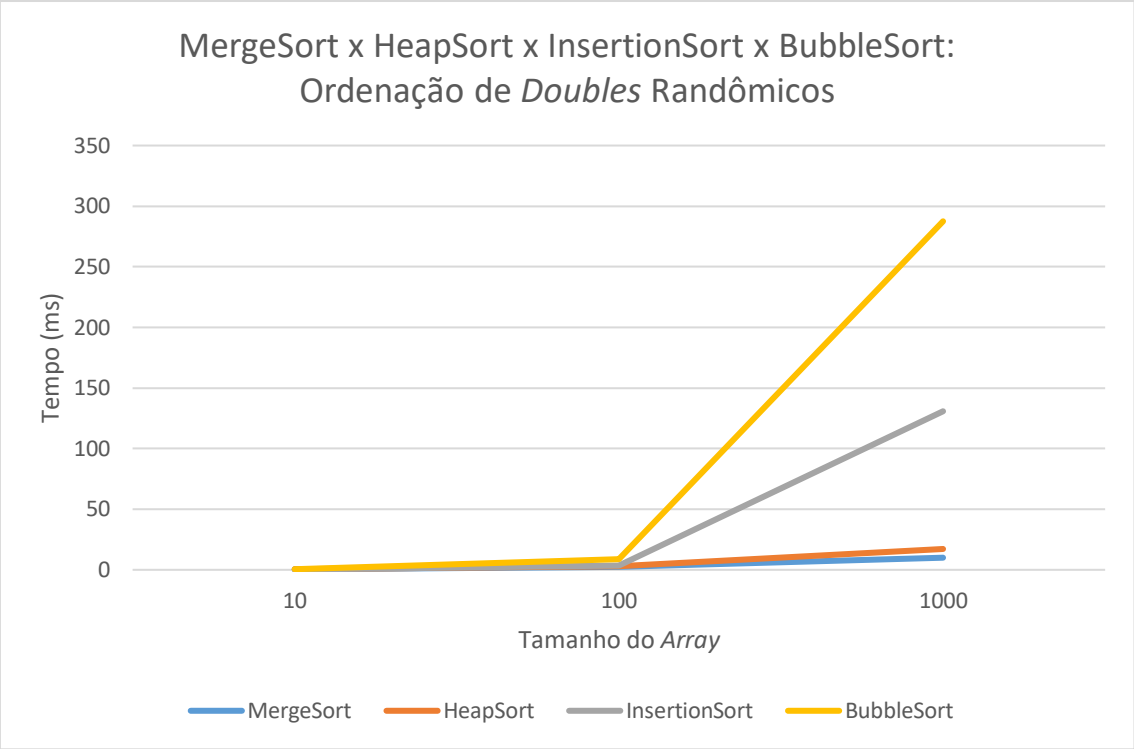




COMPARAÇÃO GERAL PARA PEQUENOS *ARRAYS*

Na próxima página, encontra-se uma comparação entre o tempo de ordenação de todos os algoritmos para *arrays* pequenos (até mil de tamanho). Podemos ver a superioridade dos algoritmos com complexidade *nlogn* mesmo em casos pequenos e concluir que, como regra geral, o *Merge Sort* é mais eficiente que o *Heap Sort*, que ordena mais rapidamente que o *Insertion Sort*, que é mais otimizado que o *Bubble Sort*. Assim, percorremos, neste relatório, os quatro algoritmos do mais eficiente para o menos eficiente.

Uma observação importante é que a medida de tempo do gráfico da próxima página é em **milissegundos, excepcionalmente**.



CONCLUSÃO GERAL

Neste relatório, o leitor foi capaz de ver os resultados dos quatro algoritmos de ordenação testados partindo do mais eficiente ao menos eficiente e de analisar os pontos positivos de cada um deles. Ficou claro a diferença que a complexidade do algoritmo faz no tempo de ordenação: pegando o caso de um array com cem mil elementos, enquanto os algoritmos *nlogn* tiveram tempos menores que cinco segundos, os n^2 tiveram tempos maiores que cento e cinquenta segundos.

Foi possível, também, ver algo interessante e que ocorreu em todos os algoritmos: ordenar tipos de dados mais complexos (como double) é mais difícil e mais lento. Todos os algoritmos tiveram tempos de ordenação maiores ao ordenar *float* e *double* em comparação com inteiros.