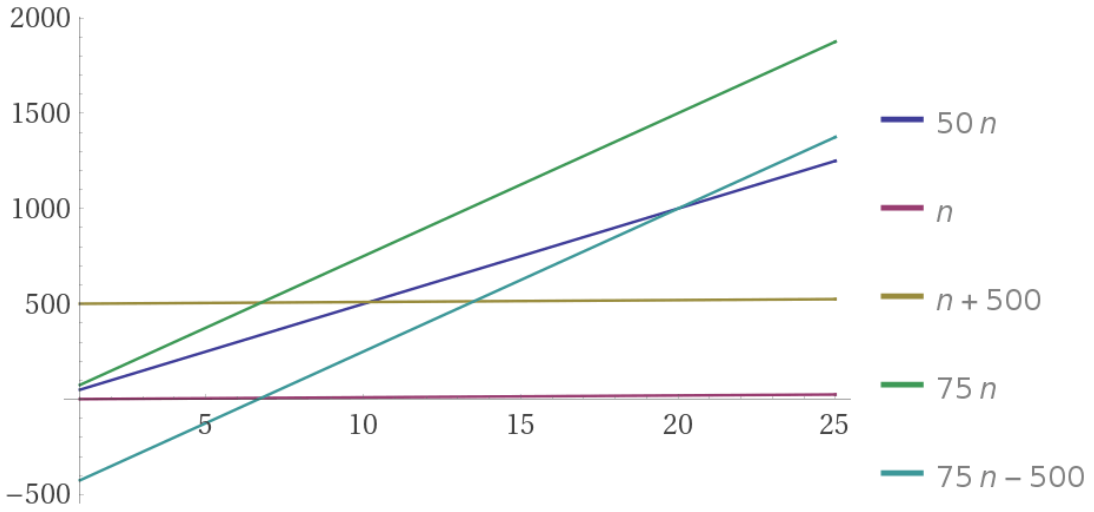
Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01124 - Classificação e Pesquisa de Dados - Turma B

Alexandre Lima – 273164 **Data: 03/09/18**

1) Complexidade de Funções:

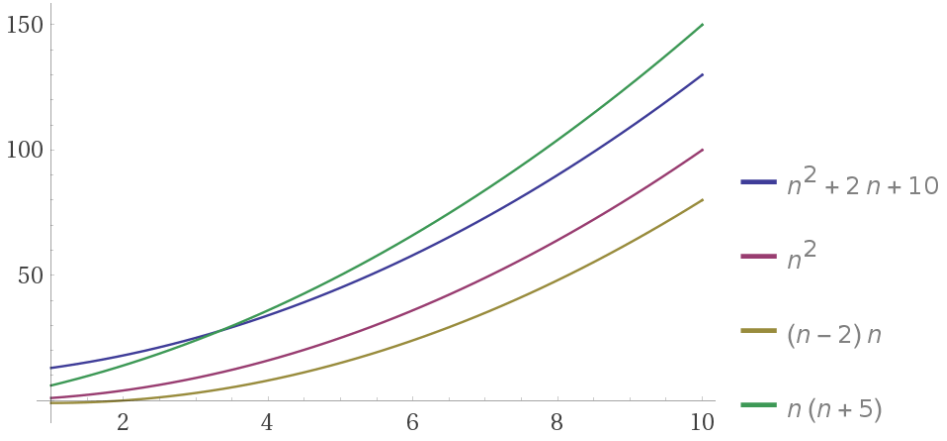
Os seguintes plots mostram que, para as funções dadas, a partir de um ponto n_0, existem outras funções onde o valor de f(n) está entre os valores destas funções:

a) $50n$:



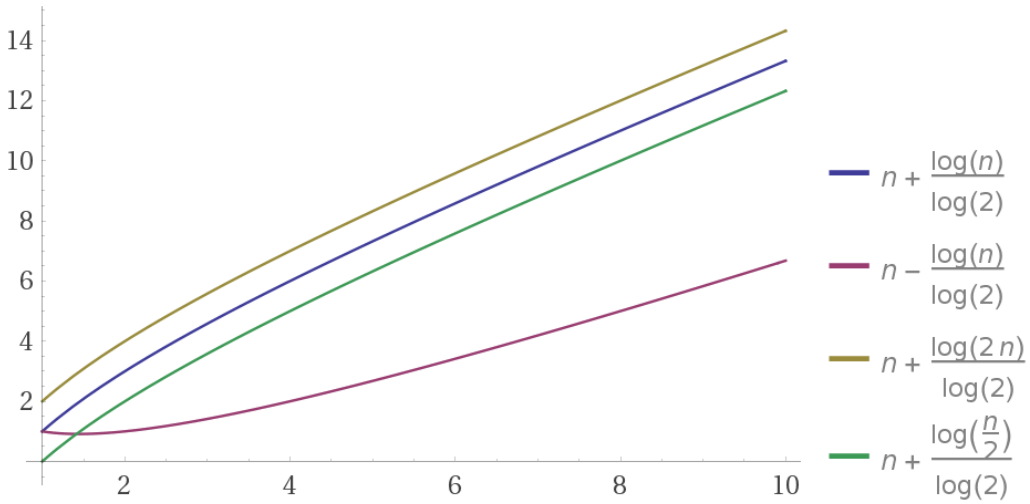
A função $50n$ é sempre limitada, para n ≥ 1, superiormente por $75n$ e inferiormente por n. Para n > 10, a função $50n$ é maior que $n + 500$, e para n > 20, a função $75n - 500$ é maior que $50n$. Ou seja, n_0 = 20, e demostra que $\Theta(50n)$.

b) $n^2 + 2n + 10$:



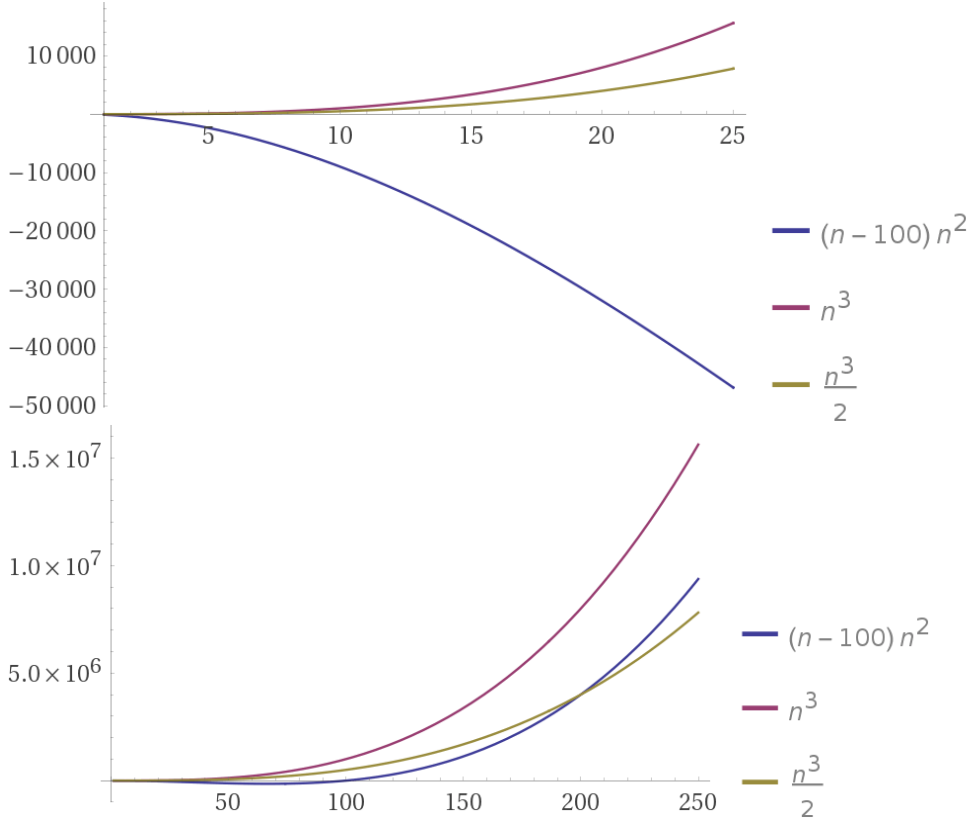
A função $n^2 + 2n + 10$ é sempre limitada, para n ≥ 1, inferiormente por n^2 e $n^2 - 2n$ Para, aproximadamente, n > 4, a função $n^2 + 5n$ é maior que $n^2 + 2n + 10$. Ou seja, neste caso, $n_0 \approx 4$, e, para n tendendo ao infinito, $n^2 + 2n + 10 = n^2$, pois a parcela n^2 é muito maior que $2n + 10$, e demostra que $\Theta(n^2)$.

c) $n + lg(n)$, onde lg(n) = $\log_2(n)$:



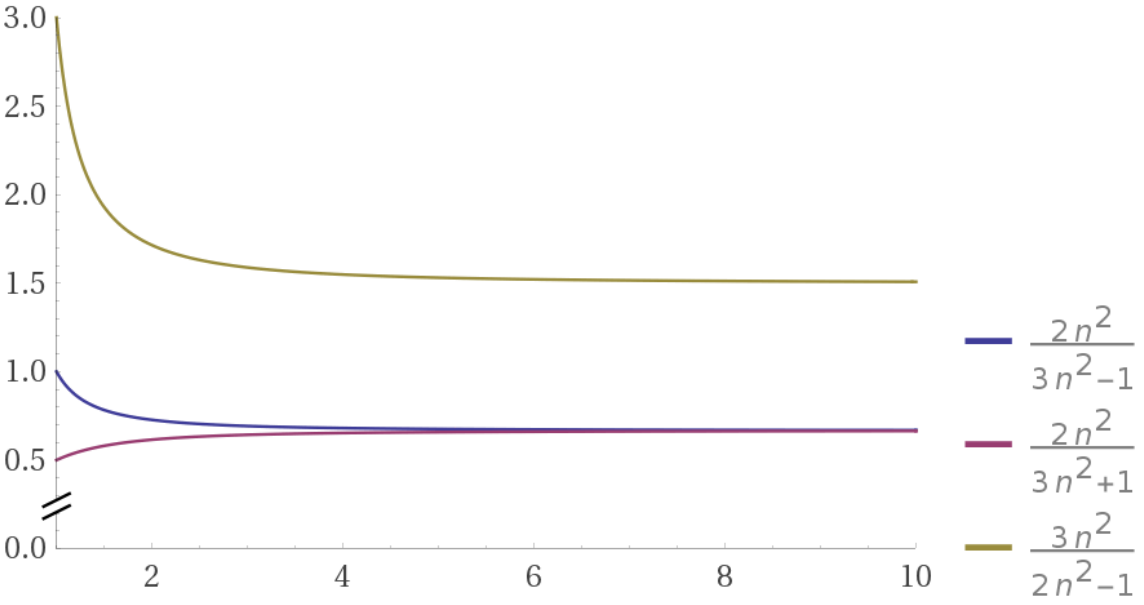
A função $n + lg(n)$ é sempre limitada, para n ≥ 1, superiormente por $n + lg(2n)$ e inferiormente por $n - lg(n)$ e $n + lg\left(\frac{n}{2}\right)$. Para n tendendo ao infinito, a parcela $lg(n)$ é insignificante, ou seja, $\Theta(n)$.

d) $n^3 - 100n^2$:



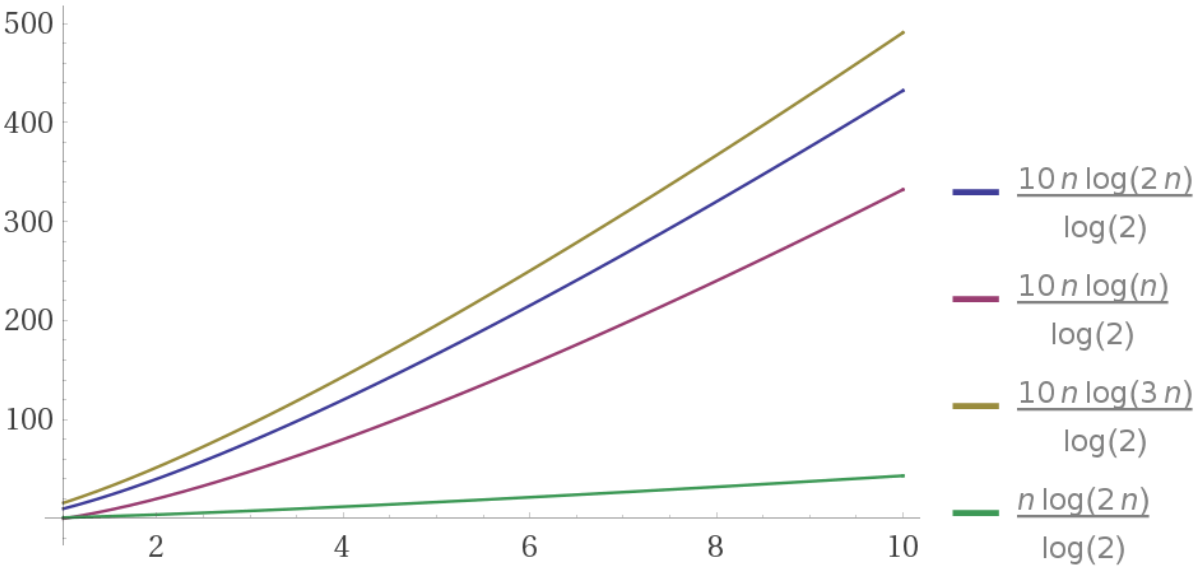
A função $n^3 - 100n^2$ é sempre limitada, para n ≥ 1, superiormente por n^3. Por ser uma função cúbica com o termo $-100n^2$, ela decresce antes de atingir o ponto de inflexão, onde então começa a crescer. Ela ultrapassa a função $\frac{n^3}{2}$ quando n = 200. Ou seja, n_0 = 200 e $\Theta(n^3)$ (pois qualquer termo de menor ordem é insignificante).

e) $\frac{2n^2}{3n^2-1}$:



A função $\frac{2n^2}{3n^2-1}$ é sempre limitada, para n ≥ 1, superiormente por $\frac{3n^2}{2n^2-1}$. A função $\frac{2n^2}{3n^2+1}$ é sempre menor ou muito próximo à função $\frac{2n^2}{3n^2-1}$, pois quando n tende ao infinito, ambas as funções tendem a $\frac{2}{3}$. Portanto, $\Theta(\frac{2}{3})$.

f) $10\ n\ lg(2n)$:



A função $10\,n\,lg(2n)$ é sempre limitada, para n ≥ 1, superiormente por $10\,n\,lg(3n)$ e inferiormente por $10\,n\,lg(n)$ e $n\,lg(2n)$. As funções $10\,n\,lg(2n)$ e $10\,n\,lg(n)$ possuem valores parecidos para valores pequenos, mas se distanciam quando n tende ao infinito, portanto, $\Theta(10\,n\,lg(2n))$.

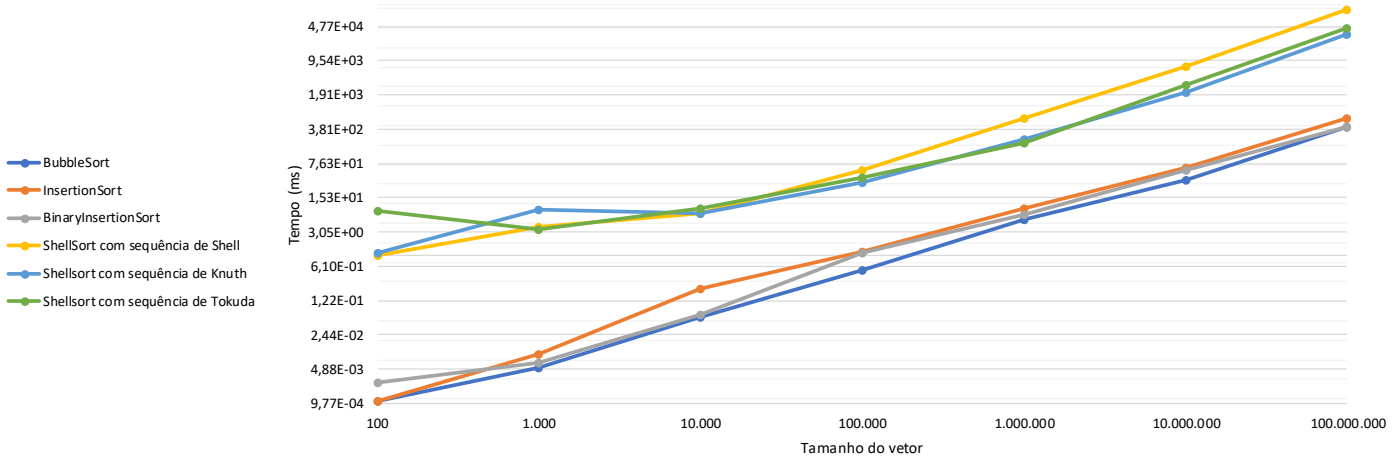
2.2) Teste dos métodos implementados com diferentes configurações de entrada:

O programa para ordenar os vetores foi feito em C# e está sendo enviado junto com o pdf.

Análise do tempo de execução dos métodos de ordenação no vetor ordenado (melhor caso)

Método de ordenação	Número de elementos no vetor						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
BubbleSort	1,1 µs	5,2 µs	57,7 µs	520,3 µs	5,45 ms	34,88 ms	439,92 ms
InsertionSort	1,1 µs	10,1 µs	209,1 µs	1,23 ms	9,26 ms	64,13 ms	638,83 ms
BinaryInsertionSort	2,6 µs	6,7 µs	63,4 µs	1,15 ms	6,94 ms	56,62 ms	427,29 ms
ShellSort com sequência de Shell	1,05 ms	3,84 ms	7,41 ms	55,78 ms	635,71 ms	7,24 s	106,09 s
Shellsort com sequência de Knuth	1,12 ms	8,57 ms	7,39 ms	30,96 ms	243,28 ms	2,23 s	33,96 s
Shellsort com sequência de Tokuda	8,17 ms	3,57 ms	9,19 ms	40,45 ms	197,83 ms	3,00 s	45,40 s

Nenhuma troca é executada nos algoritmos de ordenação quando o vetor já está ordenado

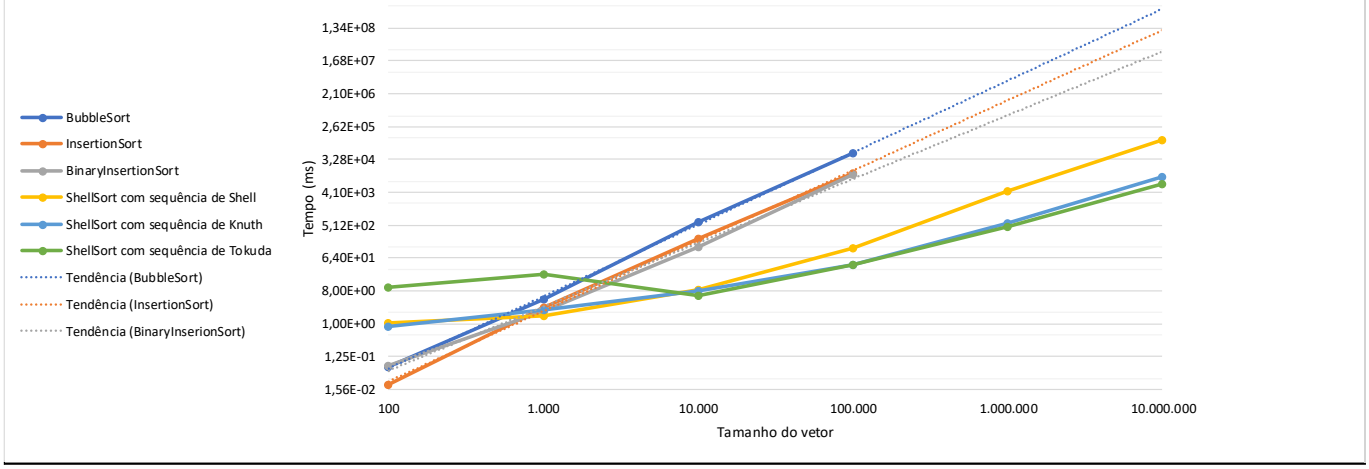


O tempo de execução dos métodos quando o vetor já está ordenado é mostrado acima. Podemos ver que desde n = 100 até n = 10.000.000, o Bubble Sort foi o mais rápido. Em n = 100.000.000, o Insertion Sort com busca binária foi um pouco mais rápido, mas não refuta a conclusão de que, neste caso, o Bubble Sort foi o método de ordenamento mais rápido. Por outro lado, o Shell Sort, mesmo com três sequências diferentes, sempre foi mais lento. Isso é comprovado pelo fato de que, para o Bubble e Insertion Sort, Ω(n), enquanto para o Shell Sort, Ω(n lg n). O Insertion Short com busca binária deveria se comportar como o Shell Sort, mas isso não aconteceu neste caso (provável erro de programação, apesar dos tempos de execução geralmente menores que o Insertion Sort). Ou seja, com o vetor já ordenado e considerando uma média do tempo de execução, os métodos se classificam:

1º - Bubble Sort
2º - Insertion Sort com busca binária
3º - Insertion Sort
4º - Shell Sort com sequência de Knuth e Tokuda
5º - Shell Sort com sequência Shell.

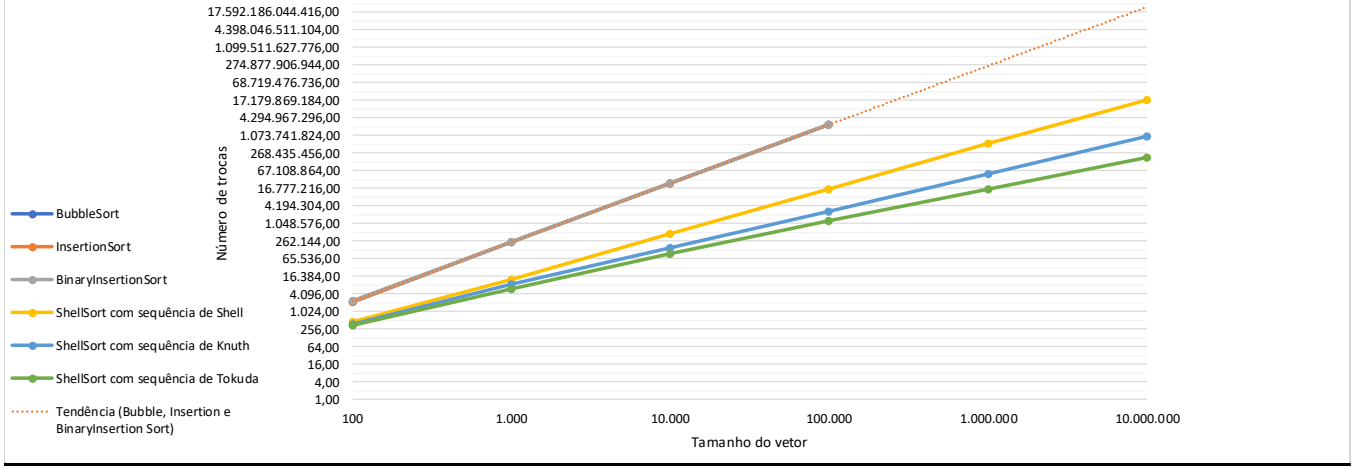
Obs: Todos os gráficos estão, no eixo Y, em escala logarítmica de ordem 2, e no eixo X, logarítmica de ordem 10.

Análise do tempo de execução dos métodos de ordenação no vetor aleatório (caso médio)							
Método de ordenação	**Número de elementos no vetor**						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
BubbleSort	63,0 µs	4,78 ms	634,67 ms	48,08 s	-	-	-
InsertionSort	21,5 µs	2,86 ms	211,53 ms	14,04 s	-	-	-
BinaryInsertionSort	67,9 µs	2,22 ms	131,21 ms	12,35 s	-	-	-
ShellSort com sequência de Shell	1,05 ms	1,60 ms	8,44 ms	120,02 ms	4,29 s	114,99 s	-
ShellSort com sequência de Knuth	852,5 µs	2,35 ms	7,86 ms	40,77 ms	600,22 ms	11,06 s	-
ShellSort com sequência de Tokuda	10,28 ms	23,62 ms	6,05 ms	42,62 ms	479,72 ms	7,02 s	-



Com o vetor aleatório, o campeão foi o Shell Sort utilizando a sequência de Tokuda. Os piores foram claramente o Bubble e Insertion Sort. O gráfico apresenta a tendência desses métodos, ou seja, quanto tempo demoraria, em média, para classificar um vetor aleatório de tamanhos maiores que o máximo registrado (n = 100.000). Caso fosse preciso calcular um vetor com 10.000.000 de elementos, demorariam aproximadamente 10^8 ms = 1.000.000 segundos, ou aproximadamente 16.666 minutos (12 dias!).

Análise do número de trocas necessárias para ordenar o vetor aleatório (caso médio)							
Método de ordenação	**Número de elementos no vetor**						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
BubbleSort	2.346	248.345	25.129.516	2.500.675.681	-	-	-
InsertionSort	2.346	248.345	25.129.516	2.500.675.681	-	-	-
BinaryInsertionSort	2.388	248.750	25.133.796	2.500.718.020	-	-	-
ShellSort com sequência de Shell	466	12.869	457.159	16.728.947	588.054.857	17.501.823.015	-
ShellSort com sequência de Knuth	383	8.592	161.708	2.796.210	51.614.537	1.006.121.220	-
ShellSort com sequência de Tokuda	348	6.249	95.785	1.303.370	16.643.346	202.895.848	-

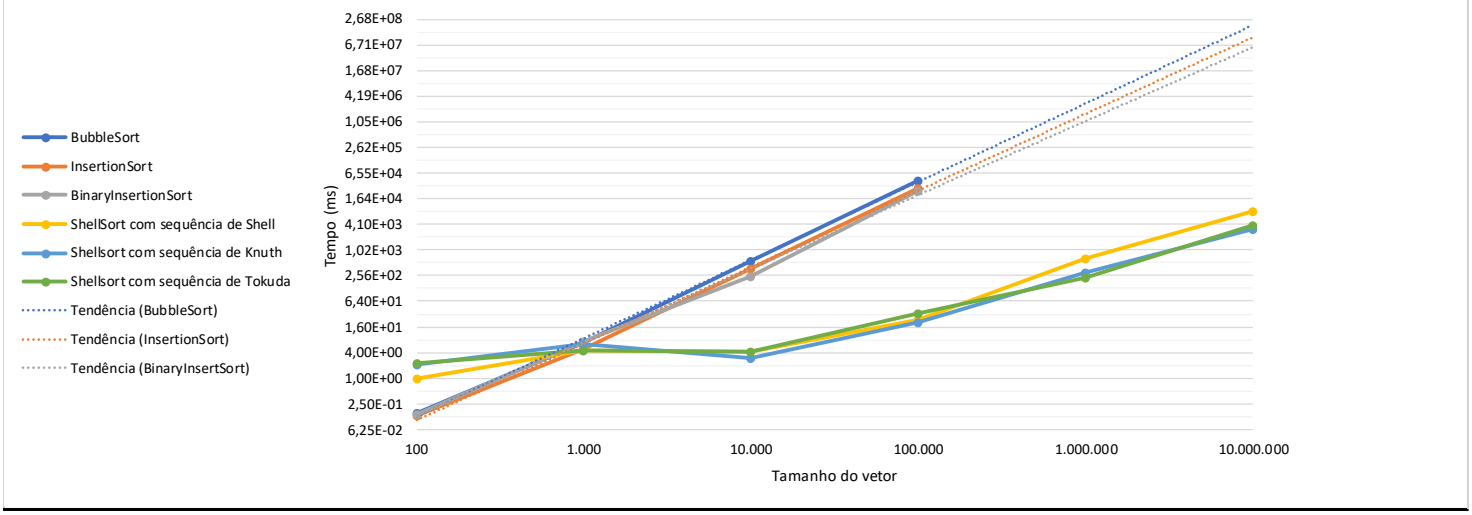


O Shell Sort também foi o método que realizou menos trocas, sendo que a sequência de Tokuda foi a que menos trocou elementos. Bubble e Insertion Sort também não foram eficientes nas trocas, tendo realizado muito mais trocas que o Shell Sort. Considerando todos esses fatos, classifico os métodos como:

1º - Shell Sort utilizando a sequência de Tokuda
2º - Shell Sort com sequência de Knuth
3º - Shell Sort com sequência de Shell
4º - Insertion Sort com busca binária
5º - Insertion Sort
6º - Bubble Sort

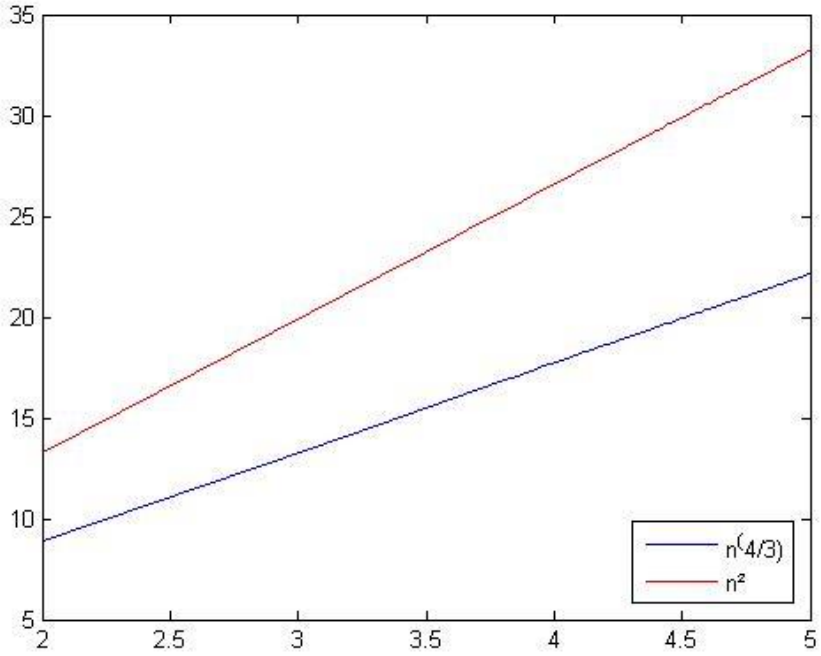
Obs: Em cada tamanho de n, foi utilizado o mesmo vetor aleatório para todos os métodos.

Análise do tempo de execução dos métodos de ordenação no vetor invertido (pior caso)							
Método de ordenação	Número de elementos no vetor						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
BubbleSort	149,5 µs	6,60 ms	572,73 ms	44,85 s	-	-	-
InsertionSort	129,2 µs	4,80 ms	372,83 ms	29,32 s	-	-	-
BinaryInsertionSort	140,8 µs	7,12 ms	248,41 ms	25,45 s	-	-	-
ShellSort com sequência de Shell	960,1 µs	4,42 ms	4,23 ms	23,20 ms	638,85 ms	8,14 s	-
Shellsort com sequência de Knuth	2,11 ms	6,41 ms	3,02 ms	20,42 ms	304,46 ms	3,19 s	-
Shellsort com sequência de Tokuda	2,27 ms	4,57 ms	4,31 ms	34,04 ms	234,00 ms	3,94 s	-



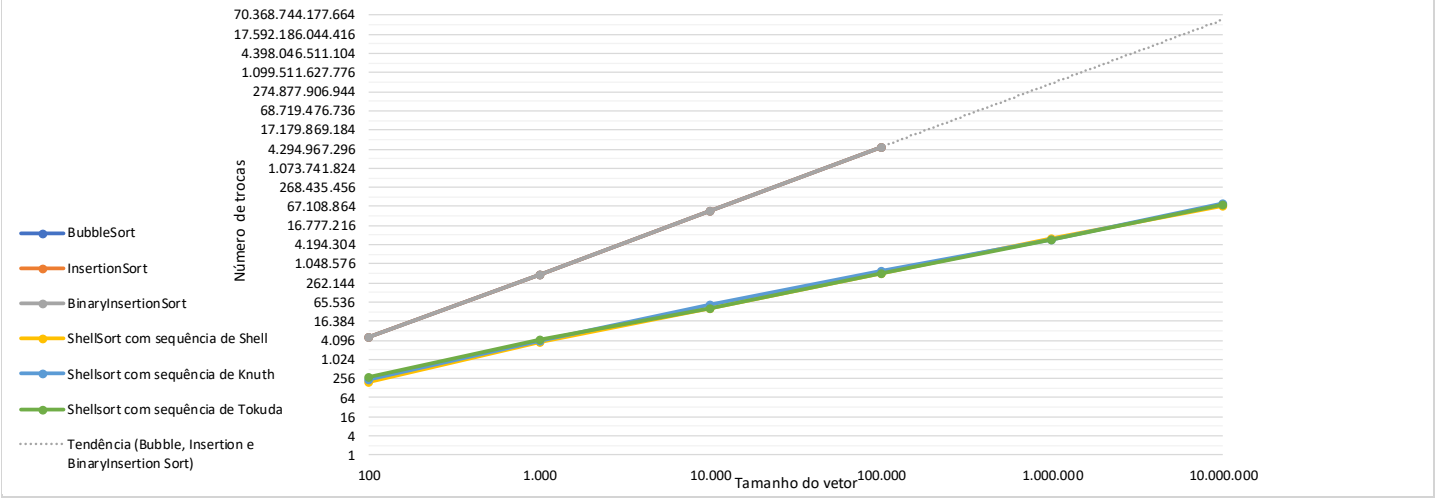
Com o vetor invertido, o caso foi semelhante ao aleatório, sendo o Shell Sort o método mais rápido, porém, tanto no aleatório como no invertido, percebo que Bubble e Insertion Sort foram mais rápidos quando o tamanho do vetor era de 100 elementos, e quando n = 1000, a diferença entre todos os métodos era mínima, sendo que somente a partir de 10.000 elementos podemos ver que o Shell Sort é mais rápido. Concluo que Bubble e Insertion Sort são eficientes quando o vetor é pequeno ou já está ordenado.

No pior caso, quando n tende ao infinito, temos que, para o Bubble e Insertion Sort, $O(n^2)$, e para o Shell Sort (usando a sequência Sedgewick, não mostrada aqui), $O(n^{4/3})$. O gráfico abaixo, feito no MATLAB, mostra o comportamento das funções n^2 e $n^{4/3}$ num gráfico com as mesmas ordens ($\log_2$ em Y e $\log_{10}$ em X):



Podemos perceber a semelhança com o gráfico feito experimentalmente, se considerarmos n > 10.000 e as linhas de tendência.

Análise do número de trocas necessárias para ordenar o vetor invertido (pior caso)							
Método de ordenação	**Número de elementos no vetor**						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
BubbleSort	4.950	499.500	49.995.000	4.999.950.000	-	-	-
InsertionSort	4.950	499.500	49.995.000	4.999.950.000	-	-	-
BinaryInsertionSort	4.950	499.500	49.995.000	4.999.950.000	-	-	-
ShellSort com sequência de Shell	204	3.564	42.144	512.976	6.782.784	68.632.512	-
Shellsort com sequência de Knuth	230	3.920	53.704	619.654	6.245.384	82.732.746	-
Shellsort com sequência de Tokuda	280	4.174	42.776	515.604	6.153.798	78.390.934	-



O gráfico se assemelha também com este, o gráfico das trocas feitas para ordenar o vetor. Neste gráfico podemos ver novamente que o Shell Sort realizou menos trocas, mas o número de trocas depende da sequência utilizada.

Considerando todos os dados apresentados, classifico os métodos, para ordenar um vetor invertido, da seguinte maneira:

1º - Shell Sort (todas as sequências apresentam valores muito semelhantes)

2º - Insertion Sort (com ou sem busca binária, já que o número de trocas é igual entre os dois métodos e também ao Bubble Sort)

3º - Bubble Sort (foi o mais lento nos casos de vetores maiores)

Obs: Ignorar os números marcados no gráfico feito no MATLAB, pois foi feito o cálculo de logaritmo ao plotar o gráfico. O seguinte programa foi utilizado:

```
x = linspace(100,100000);
ss = x.^(4./3);
bs = x.^2;
plot(log10(x),log2(ss),'b-');
hold on
plot(log10(x),log2(bs),'r-');
legend('n^(4/3)','n²');
```