Universidade Estadual Paulista - UNESP
Faculdade de Ciências e Tecnologia
Departamento de Matemática e Computação

Análise sobre Algoritmos de Ordenação

Projeto e Análise de Algoritmos

Danilo Medeiros Eler
Gilmar Francisco de Oliveira Santos

2018
Outubro

Sumário

1 Introdução

Os Algoritmos de ordenação tem como propósito organizar dados de entrada, geralmente um vetor, rearranjar os elementos e devolver o mesmo em uma determinada ordem, geralmente empregado para valores numéricos ou léxicos, os colocando do menor para o maior ou vice-versa, porém expansível para diversos tipo de dados.

A ordenação é muito relevante computacionalmente, visto a sua necessidade em muitos dos algoritmos de busca e dentre outras aplicações. A lista de algoritmos de ordenação é bastante extensa, mediante a existência de aplicações aos mais diversos casos, ou mesmo versões melhoradas e variações de um mesmo algoritmo.

A operações básicas presentes nesse tipo de algoritmo geralmente são a comparação de valores, cópia o swap. A **Complexidade Espacial** é um importante assunto a ser tratado sobre algoritmos de ordenação; alguns necessitam que o vetor de entrada seja copiado e outros ordenam no próprio vetor (in-place) tudo isso influência não só no tempo de execução do mesmo, mas também na quantidade de memória gasta. A **Estabilidade** é a característica onde dois objetos com as mesmas chaves aparecem na mesma ordem no vetor de entrada, e continuam na mesma ordem no vetor de saída. Neste trabalho a **Complexidade de Tempo** estará em enfoque.

2 Bubblesort Original

Algoritmo de ordenação por flutuação: o que é mais "denso" se deposita no fundo e o que é mais leve flutua. Em seus 3 casos ele executa em $O(n^2)$ [3]. É um algoritmo de fácil implementação, que pode ser utilizado para um "n" pequeno, ao passo que a sua complexidade espacial é O(1), ou seja, não utiliza mais que o espaço do próprio vetor para realizar a ordenação, porém para vetores de maior tamanho o algoritmo torna-se muito custoso, visto sua complexidade quadrática. Na análise experimental foi claramente notável a semelhança entre os 3 casos, a única diferença é um gasto menor de tempo para entradas ordenas, visto que não é necessário que os elementos sejam trocados, apenas verificações.

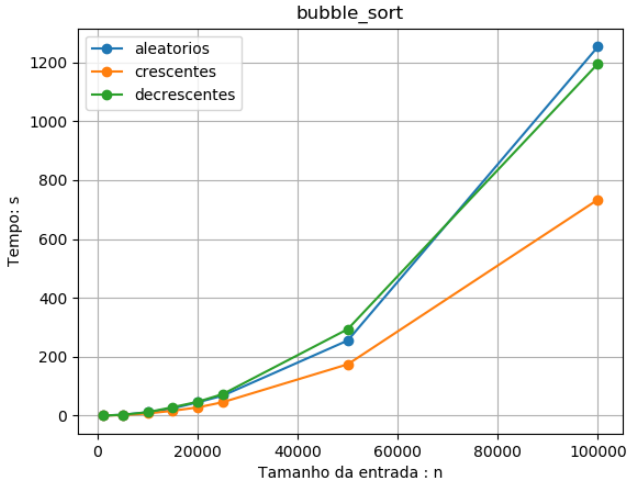

Figure 1: Resultado Bubblesort clássico

3 Bubblesort Melhorado

Adicionado a verificação se o vetor já está ordenado, nesta situação o melhor caso do algoritmo torna-se $\Omega(n)$. e os demais casos continuam sendo $O(n^2)$.

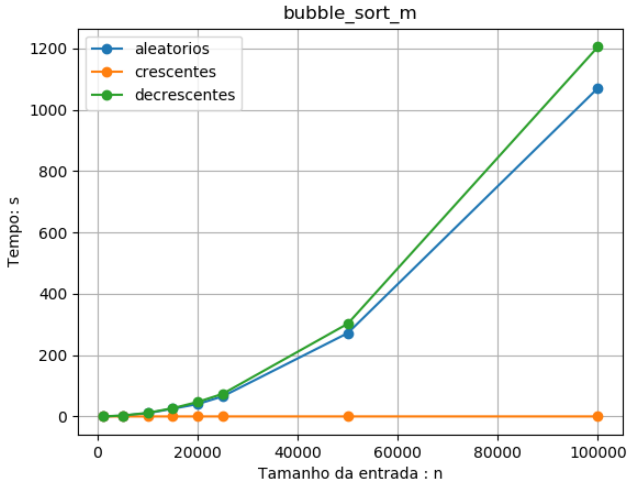

Figure 2: Resultado Bubblesort Melhorado

4 Quicksort

Algoritmo que tem base na divisão e conquista, tem o seu melhor caso sendo Ω(n log(n)), seu caso médio é Θ(nlog(n)) e seu pior caso em $O(n^2)$ [6]. É um bom algoritmo para dados aleatórios, o que explica sua grande popularidade, porém seu pior caso ocorre quando é escolhido de forma equivocada o pivô, sendo ele o maior elemento ou o menor elemento, o que acontece é que as partições ficam com tamanhos muito diferentes levando a ineficiência do algoritmo. Sua complexidade espacial é $O(\log(n))$, visto que cria partições menores do vetor original.

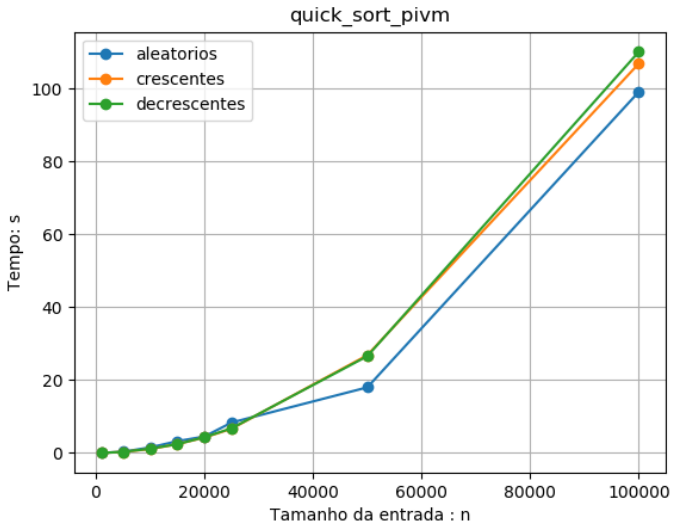

Figure 3: Resultado Quicksort pivô elemento do meio: 3 casos semelhantes

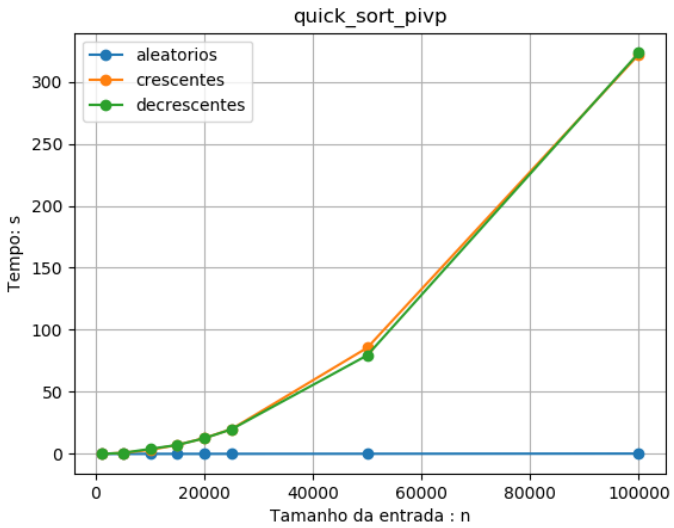

Figure 4: Resultado Quicksort pivô primeiro elemento: acontece o melhor caso, onde as partições tem tamanhos semelhantes.

5 Mergesort

É um algoritmo que utiliza do principio da divisão e conquista para realizar a ordenação. Tem o seu melhor caso Ω(nlog(n)), seu caso médio em Θ(nlog(n)) e seu pior caso em O(nlog(n)), o que o faz ser considerado estável [7]. Porém apresenta uma complexidade espacial de O(n), visto as divisão e depois união das partes (conquista) necessárias para a ordenação. Podemos observar que os seu 3 casos são iguais, o que mostra que o mergesort é não sensível a entrada [Ver abaixo], pois realiza sempre particionamentos de mesmo tamanho.

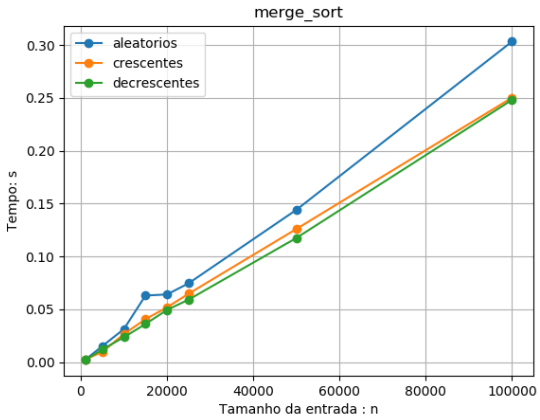

Figure 5: Resultado Mergesort: tempos de ordenação muito semelhantes para todos os casos.

6 Heapsort

Utiliza da estrutura de heap para realizar a ordenação. Apresenta melhor caso, caso médio e pior caso proporcionais a O(nlog(n)) [8], o que demonstra que o mesmo é não sensível a entrada, pois executa de maneira semelhante tanto para entradas ordenadas, quanto aleatórias [Ver abaixo]. Como realiza a ordenação utilizando o próprio vetor original, a sua complexidade espacial é de O(1).

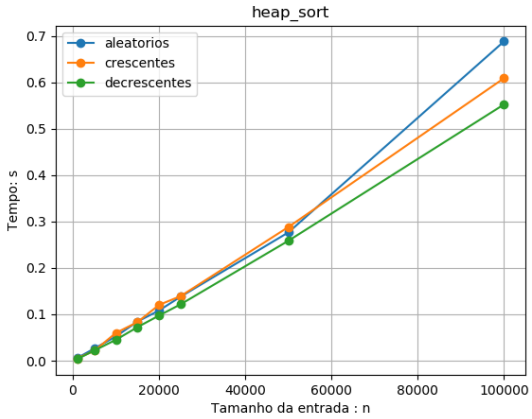

Figure 6: Resultado Heapsort

7 Insertionsort

Utiliza da ideia da inserção em uma fila ordenada. Apresenta melhor caso sendo $\Omega(n)$; o qual ocorre para vetores ordenados ou parcialmente ordenados, seu caso médio e pior caso são iguais a $O(n^2)$ [9], pois não trabalha muito bem com vetores aleatórios. Como a "inserção" é realizada no próprio vetor, a sua complexidade espacial é de $O(1)$.

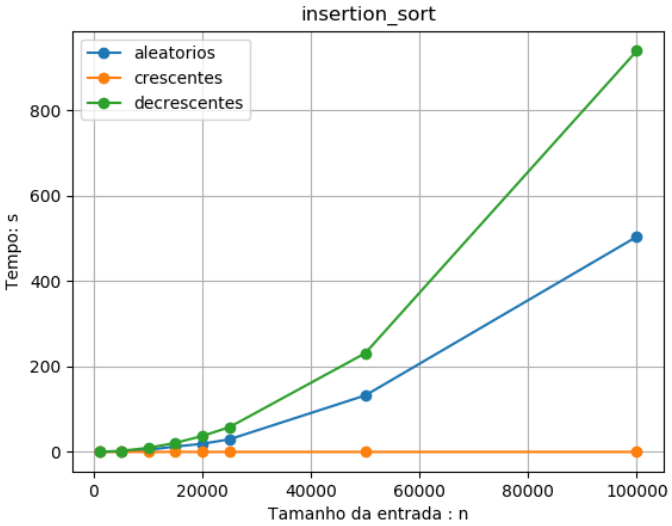

Figure 7: Resultado Insertionsort: melhor caso ocorre para entradas crescentes, enquanto o pior caso ocorre para entradas decrescentes.

8 ShellSort

É um melhoramento do Insetionsort que tem apresenta um gap maior que 1 elemento, seu melhor caso corresponde a $\Omega(n\log(n))$ e ocorre para entradas quase ordenadas, enquanto seu caso médio e pior caso são iguais $\Theta(n(\log^2(n))$ ou conforme Sedgewick $O(n^{4/3})$ no pior caso [2]. Sua complexidade espacial fica em $O(1)$. Nesse trabalho foi utilizada a sequência de Ciura, que se mostrou experimentalmente melhor que as outras sequências de gap como a de Sedgewick. [1].

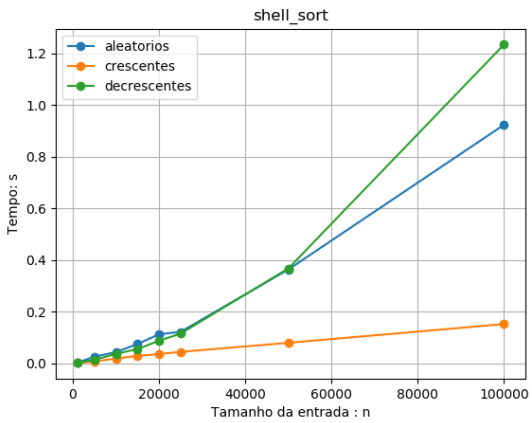

Figure 8: Resultado Shellsort para sequência de gaps [701, 301, 132, 57, 23, 10, 4, 1]

9 Selectionsort

O algoritmo utiliza o princípio de sempre encontrar o menor elemento e o colocar em sua posição adequada no vetor. Ele é semelhante ao Bubblesort sem melhoria. Sendo seus 3 casos proporcionais a $O(n^2)$ [10]. Experimentalmente ele apresenta vantagem sobre o Bubblesort. E é um algorítimo muito estável para qualquer tipo de entrada, como é possível observar abaixo:

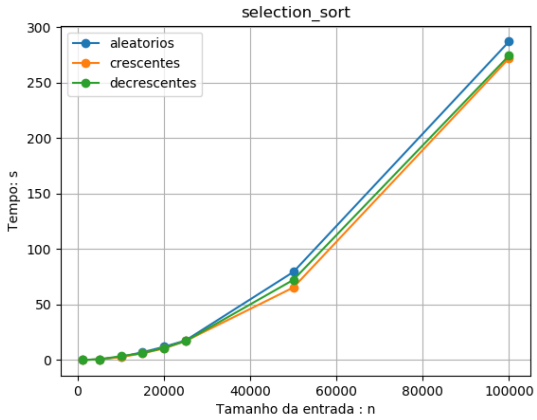

Figure 9: Resultado Selectionsort

10 Método

Utilizando o linguagem python foram gerados 3 tipos de arquivos para cada quantidade de dados, ordenados: crescente, decrescente e aleatórios, os arquivos referentes ao teste experimental podem ser baixados em `https://github.com/gilmarfrancisco828/T01_PAA`. Para o teste experimental foi feita a leitura dos inteiros salvos anteriormente nos arquivos e carregados em vetores na memória (para que a ordenação pudesse ser realizada mais rapidamente, esse processo foi realizado antes da medida do tempo para que não ocorressem interferências na análise). E cada vetor (tamanho, tipo) for submetido a ordenação pelos algoritmos decritos anteriormente e o tempo de execução medido com recisão de 15 casa decimais.

Na seção de Resultados Experimentais constam as tabelas com os tempos de ordenação, que para a melhor visualização apresentam precisão de 4 casas. Utilizando da Biblioteca matplot e os resultados obtidos, foram gerados os gráficos que constam na seção de <Resultados Experimentais>.

10.1 Hardware e Software utilizados nos testes Experimentais

- Processador: Intel Core i7-7700HQ CPU @ 2.80GHz x 8
- RAM: 16GB Kingston
- SO: Ubuntu 16.04.5 LTS 64-bit Gnome3
- Editor de Texto: VS Code v1.28.1
- Linguagem: Python 3.5.2 64-bit
- Bibliotecas: Numpy v1.15.1, Scypy v1.1.0, matplotlib v3.0.0.

11 Resultados Experimentais

Para o catálogo e geração de gráficos foi utiliza a biblioteca matplot do python (Ver descrição de Hardware e Software). Os arquivos texto com os resultados estão salvos na pasta de "Resultados", presente no repositório.

11.1 Entradas Aleatórias

Para entrada de dados aleatória é facilmente notável que algoritmos como o Quicksort, ShellSort ou outros algoritmos baseados em divisão e conquista e particionamento são os mais indicados:

Algoritmo	**Número de elementos do vetor**							
	1000	**5000**	**10000**	**15000**	**20000**	**25000**	**50000**	**100000**
bubbleSort	0.1205	2.6833	11.3439	24.6648	45.0675	67.7314	255.3029	1253.3219
bubbleSortM	0.1027	2.5444	10.5627	25.2195	40.2927	65.4530	271.7309	1070.5516
heapSort	0.0057	0.0263	0.0523	0.0843	0.1081	0.1379	0.2765	0.6880
insertionSort	0.0594	1.3235	4.9474	12.4794	19.0760	29.5371	132.2940	503.8637
MergeSort	0.0021	0.0152	0.0313	0.0630	0.0640	0.0746	0.1441	0.3034
QuickSortPivM	0.0080	0.3138	1.5135	3.2253	4.5025	8.3376	17.989	99.0538
QuickSortPivP	0.0017	0.0168	0.0231	0.0369	0.0408	0.0627	0.1124	0.2402
SelectionSort	0.0339	0.7289	2.8884	7.0576	11.930	17.681	79.410	287.0386
ShellSort	0.0026	0.0257	0.0435	0.0747	0.1126	0.1219	0.3636	0.9235

Table 1: Resultados em segundos dos algoritmos de ordenação para dados aleatórios

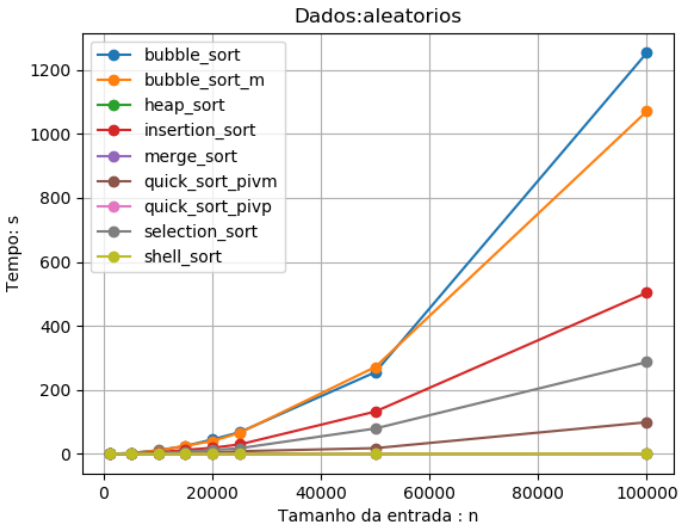

Figure 10: Resultado para entrada de dados aleatória

11.2 Entradas Ordenadas Crescentemente

Entradas crescentes geralmente geram melhores casos para algoritmos muito sensíveis a entrada, como o Bubblesort Melhorado e insertionSort.

Algoritmo	Número de elementos do vetor							
	1000	**5000**	**10000**	**15000**	**20000**	**25000**	**50000**	**100000**
bubbleSort	0.0920	1.8210	7.3573	16.417	27.8725	45.5481	174.3917	733.9344
bubbleSortM	0.0001	0.0004	0.0007	0.001	0.0015	0.0020	0.0037	0.0134
heapSort	0.0050	0.0215	0.0598	0.083	0.1202	0.1391	0.2880	0.6086
insertionSort	0.0001	0.0005	0.0009	0.001	0.0021	0.0026	0.0050	0.0155
MergeSort	0.0024	0.0096	0.0266	0.040	0.0518	0.0647	0.1260	0.2502
QuickSortPivM	0.0177	0.2748	1.0689	2.403	4.2333	6.6343	26.8120	106.8023
QuickSortPivP	0.0360	0.8027	3.2692	7.230	12.7422	20.0302	85.5832	321.9686
SelectionSort	0.0277	0.7007	2.6305	6.094	10.4270	17.1721	65.4112	271.9045
ShellSort	0.0013	0.0072	0.0189	0.028	0.0360	0.0444	0.0795	0.1520

Table 2: Resultados em segundos dos algoritmos de ordenação para dados crescentes

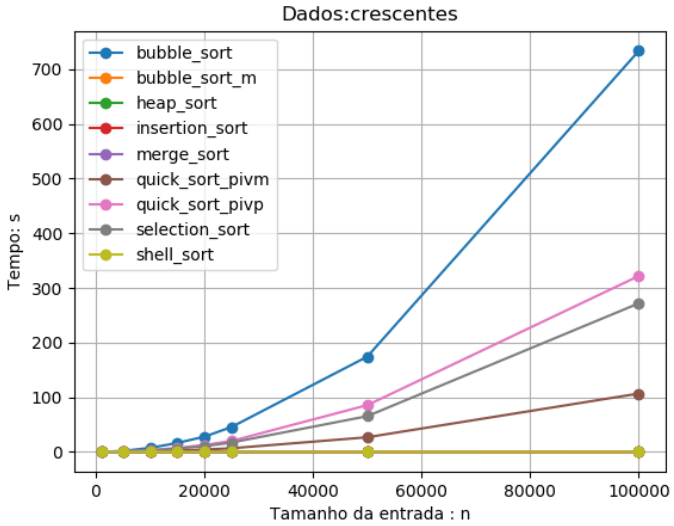

Figure 11: Resultado para entrada de dados crescente

11.3 Entradas Ordenadas Decrescentemente

Entradas ordenadas de maneira descrescente fazem com que vários dos algoritmos atinjam seus piores casos, o que é evidenciado pela elevação do teto para 1200s.

Algoritmo	Número de elementos do vetor							
	1000	**5000**	**10000**	**15000**	**20000**	**25000**	**50000**	**100000**
bubbleSort	0.1575	2.9523	11.5620	28.5389	46.9204	73.0083	293.8050	1195.6926
bubbleSortM	0.1409	2.9037	11.6687	26.7478	47.2076	73.7579	302.4040	1205.0179
heapSort	0.0033	0.0220	0.0450	0.0723	0.0973	0.1215	0.2584	0.5521
insertionSort	0.1078	2.3851	9.4417	21.0452	37.3525	58.1049	231.2657	939.0387
MergeSort	0.0020	0.0119	0.0236	0.0360	0.0494	0.0591	0.1175	0.2482
QuickSortPivM	0.0151	0.2881	1.1517	2.4017	4.3101	6.7552	26.5462	110.0228
QuickSortPivP	0.0402	0.8008	4.0329	7.1814	12.6456	19.8759	79.5031	323.2435
SelectionSort	0.0332	0.7067	3.4609	6.2206	10.9902	17.0810	72.2877	274.3372
ShellSort	0.0019	0.0134	0.0366	0.0551	0.0877	0.1152	0.3674	1.2341

Table 3: Resultados em segundos dos algoritmos de ordenação para dados descrescentes

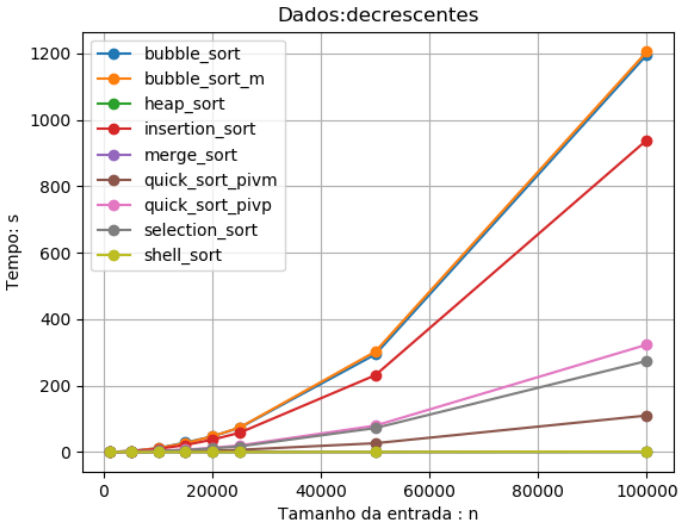

Figure 12: Resultado para entrada de dados descrescente

12 Conclusão

O melhor algoritmo para casos genéricos; vetores aleatórios e de maiores tamanhos, ou mesmo entradas ordenadas se mostrou experimentalmente ser o ShellSort, neste caso utilizando a sequência de Ciura, porém se faz necessário analisar com extremo cuidado cada caso em que se trabalha, para verificar se existe algum algoritmo de ordenação que faz mais sentido a situação problema, sempre tendo em mente o uso balanceado dos recursos, para maximizar a eficência dos sistemas computacionais.

Em contra ponto o Bubblesort original tem um custo maior até mesmo que os algoritmos com mesma complexidade como o Insertionsort, seu uso deve ser evitado sempre que possível.

Referências

[1] Marcin Ciura, *Best Increments for the Average Case of Shellsort*, 2001.

[2] Sedgewick, R., *Analysis of Shellsort and Related Algorithms*, 1996.

[3] Owen Astrachan, *Bubble Sort: An Archaeological Algorithmic Analysis*, 2003.

[4] D. A. Bell, *The Principles of Sorting*, 1958.

[5] Joshua Knowles, *Sorting Algorithms: Correctness, Complexity and Other Properties*, 2013.

[6] C. A. R. Hoare, *Quicksort*, 1962.

[7] Song Qin, *Merge Sort Algorithm*

[8] Schaffer R, Sedgewick R., *The Analysis of Heapsort*, 2002.

[9] Tarundeep S. S., Surmeet K., Snehdeep K., *Enhanced Insertion Sort Algorithm*, 2013.

[10] Sunita C., Teshu C., Rubina P, *Upgraded Selection Sort*, 2011.