UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

Análise Experimental de Algoritmos de Ordenação

David Jr. Rodrigues

Gabriel Cecon Carlsen

Pedro Henrique Zago Costa

PRESIDENTE PRUDENTE

2021

Resumo Introdutório

Este relatório tem como finalidade o estudo e implementação dos algoritmos de ordenação: BubbleSort (versão original), BubbleSort (versão melhorada), QuickSort (pivô no início da lista), QuickSort (pivô no centro da lista), InsertionSort, ShellSort, SelectionSort, HeapSort e MergeSort. Seguidamente da comparação de dados provenientes da análise experimental e assintótica (método de descrever o comportamento de limites).

Tais funções de ordenação foram desenvolvidas na liguagem de programação de alto nível Python (versão 3), executando cada algoritmo com multiplas entradas de valores (1000, 5000, 10000, 15000 e 25000), assim como três diferentes modos de dados: valores aleátorios, ordenados crescentemente e decrescentemente.

Introdução Teórica

BubbleSort

A ideia básica do algoritmo é percorrer o vetor várias vezes, e a cada passagem ir comparando elementos adjacentes, trocando-os de posição caso estejam fora de ordem, assim o maior elemento “flutua ao topo” (final do vetor). Essa movimentação assemelha-se com a forma que as bolhas em um tanque de água procuram seu próprio nível. Pelo fato de ser simples o BubbleSort é altamente usado para a introdução do conceito de algoritmos de ordenação.

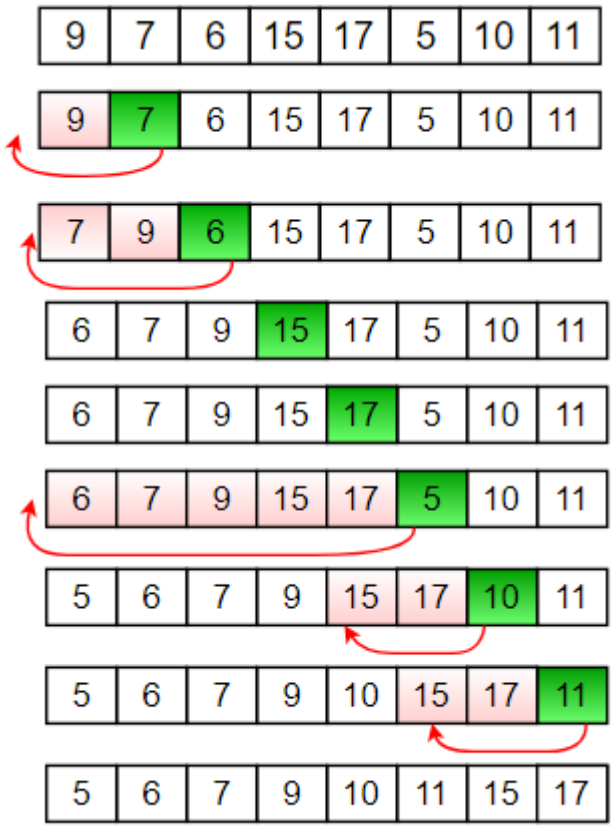
QuickSort

Desenvolvido na década de 60 o algoritmo QuickSort baseia-se na estratégia de “Divisão e Conquista”, onde dividimos o vetor inicial em varios subvetores que serão ordenados de forma independente (Divisão), sucessivamente os combinamos para produzir o resultado (Conquista). Abaixo podemos ver o passo a passo:

1. Definir a posição do pivô
2. Particionar o vetor em duas partes, onde o subvetor a esquerda contém todos os valores menores que o pivô e a direita todos os valores maiores que o pivô.
3. Ordenar os subvetores e fazer a junção, gerando um novo vetor mais ordenado que o inicial.

InsertionSort

Algoritmo que define percorrer as posições do vetor começando pelo primeiro índice. Cada nova posição é como a nova carta que você recebeu, e precisa inseri-la no lugar correto no subvetor ordenado à esquerda daquela posição. Exemplo abaixo.

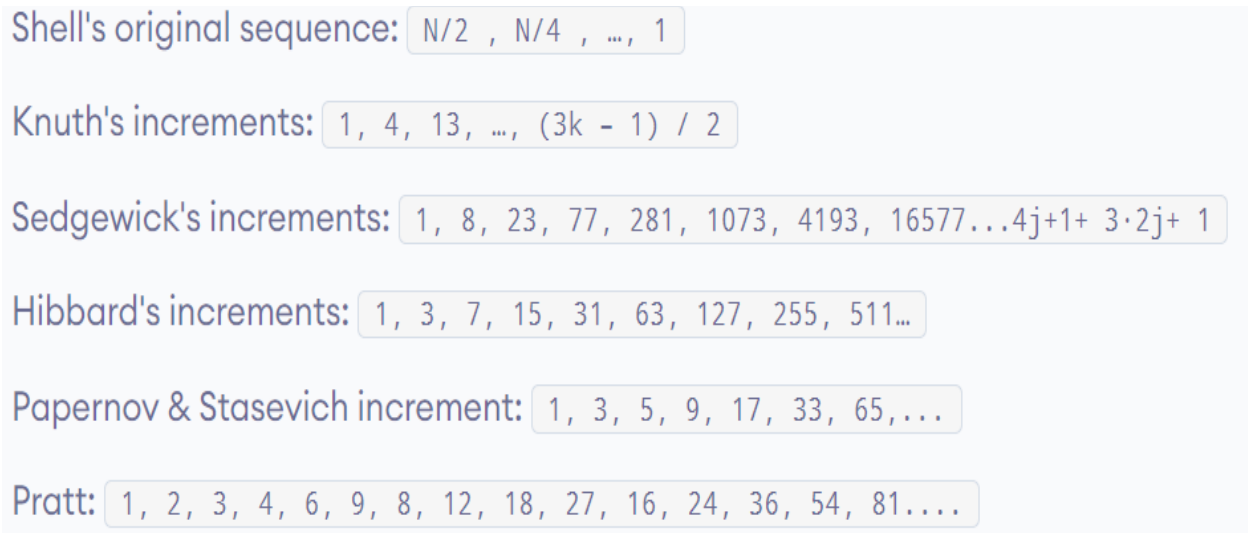


A cada nova “carta” verde temos que ordenar em relação ao subvetor vermelho a esquerda. Utilizamos o mesmo método para as próximas cartas até que o vetor original esteja completamente ordenado.

ShellSort

Utiliza a quebra sucessiva da sequência a ser ordenada. Caracterizado como melhoria do algoritmo de inserção direta, se difere pelo fato de considerar vários K subconjuntos do vetor original e a partir deles aplicar a inserção direta, sendo que K é reduzido gradativamente, assim a cada nova iteração o vetor original está mais ordenado. Então basicamente o algoritmo passa várias vezes pelo vetor dividindo o grupo maior em menores. Nos grupos menores é aplicado o método de inserção direta.

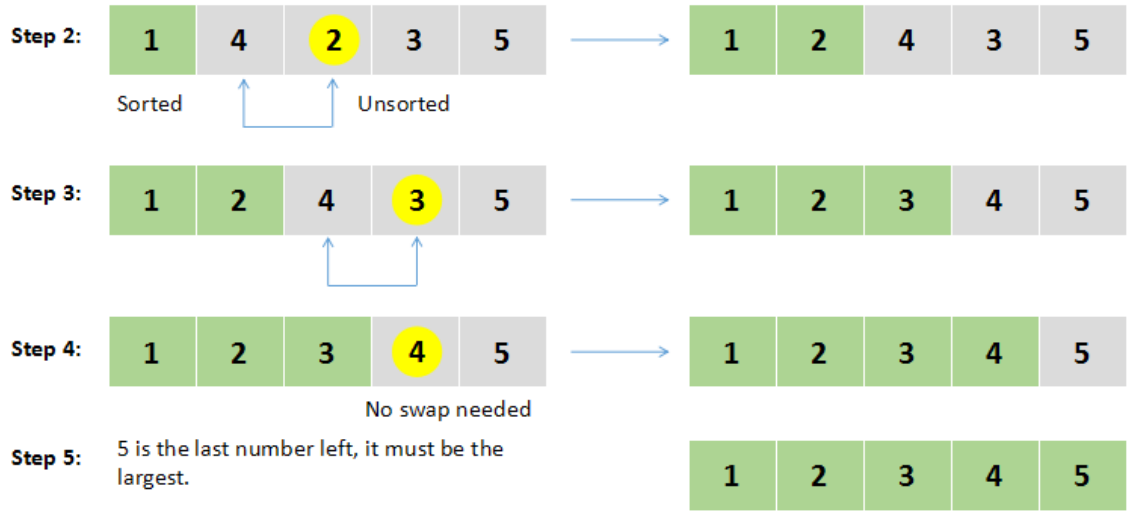
Tais grupos menores (intervalos) são decididos segundo uma sequência especifica, as mais conhecidas são os incrementos: Knuth, Sedgewick, Hibbard, Papernov & Stasevich e Pratt.



SelectionSort

Considerado um dos algoritmos de ordenação mais simples, baseia-se em passar sempre o menor valor do vetor para a primeira posição, depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os n-1 elementos restantes, até os últimos dois elementos.

1. Selecionar o menor elemento do vetor.
2. Trocar o elemento com primeiro da sequência, vetor[0].
3. Repetir os passos 1 e 2 evolvendo apenas os n-1, n-2 … elementos restantes, até restar um elemento no vetor (o maior).

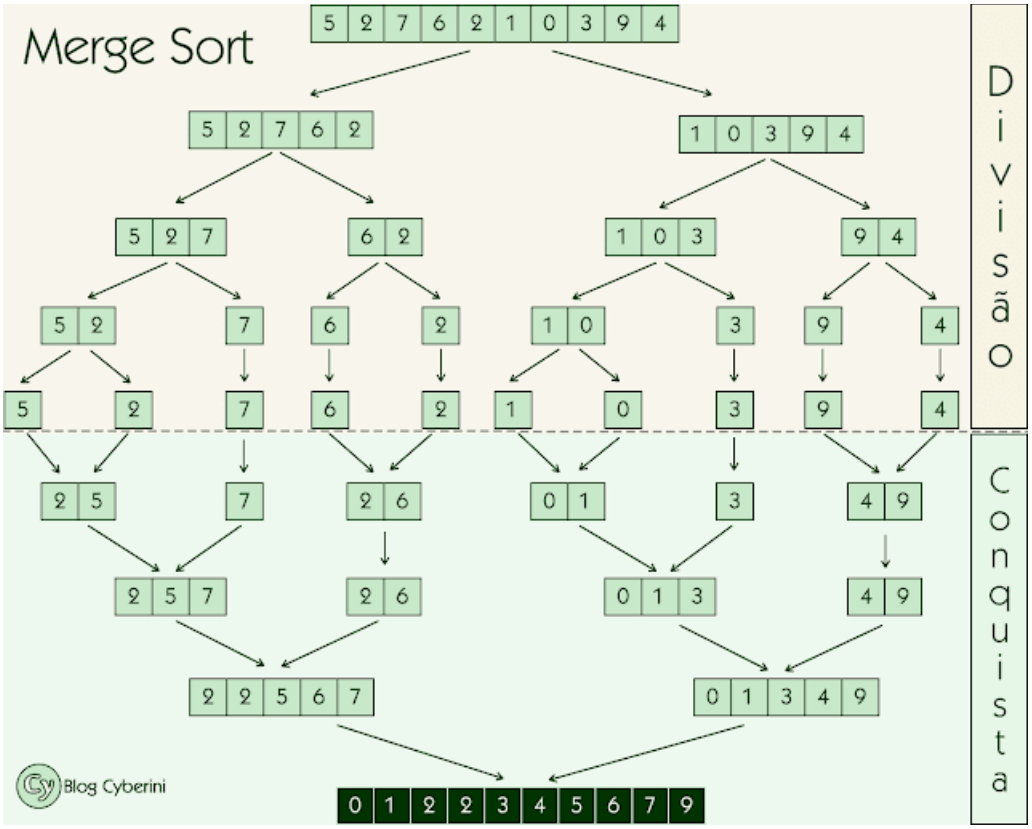


HeapSort

HeapSort usa método de seleção que ordena através de sucessivas seleções do elemento correto a ser posicionado em um segmento ordenado. Utiliza uma estrutura Heap (exerga o vetor como árvore binária) para manter na raiz o próximo elemento a ser selecionado. Segue conceitos de ordem, como o Heap máximo sendo o pai maior ou igual aos filhos (raiz o maior elemento) e o Heap mínimo, com o pai menor ou igual aos filhos, tal que a raiz é o menor elemento. Além disso, segue o conceito de forma, representando uma árvore binária que tem seus nós-folhas, em no máximo dois níveis e com as folhas localizadas ao lado esquerdo. A ideia para ordenar é construir o Heap máximo, trocar a raiz com o elemento da última posição do vetor, diminuir o tamanho do Heap, caso necessário reorganizar o Heap novamente e por fim, repetir o processo n-1 vezes.

MergeSort

Algoritmo de ordenação por comparação que usa a estratégia “Divisão e Conquista” como o QuickSort. Sua ideia básica consiste em Dividir (o vetor em vários subvetores e resolver esses subvetores através da recursividade) e Conquistar (após todos os subvetores terem sido resolvidos ocorre a conquista que é a união das resoluções). O uso da recursividade em seu algoritmo resulta em alto consumo de memória e tempo de execução. Abaixo temos de forma clara os processos de divisão e conquista.



Material e Métodos

Material

O hardware é altamente impactante nos tempos de execução de um algoritmo de ordenação em uma análise experimental, logo utilizamos a mesma configuração para todos os testes. Sua especificações são:

Processador: AMD Ryzen 1400 Quad-Core 3.4 GHz, temperatura durante os testes: aproximadamente 70 ºC;
Placa de vídeo: Sapphire Radeon RX 580 4GB GDDR5;
Memória RAM: 16 GB;

Para a execução do programa utilizamos o software de desenvolvimento **"Visual Studio Code"** (versão 1.53), com a extensão Python, **"Infogram"** para a geração de gráficos e o **"LibreOffice Calc"** para tabelas de tempo.

Métodos

Para a análise dos algoritmos usamos diversas entradas, ou seja, tamanhos de vetores, sendo eles 1000, 5000, 10000, 15000 e 25000. E para cada valor de entrada três modos: valores aleatórios (modo 1), valores ordenados crescentemente (modo 2) e decrescentemente (modo 3).

Com relação a medição de tempo de duração dos algoritmos de ordenação, fizemos a importação da biblioteca time (import time). Após o usuário escolher o número de entradas e o modo dos valores, um contador **clock_start** recebe o método **time.time()**, ou seja, **clock_start = time.time()** e ao final da ordenação uma segunda

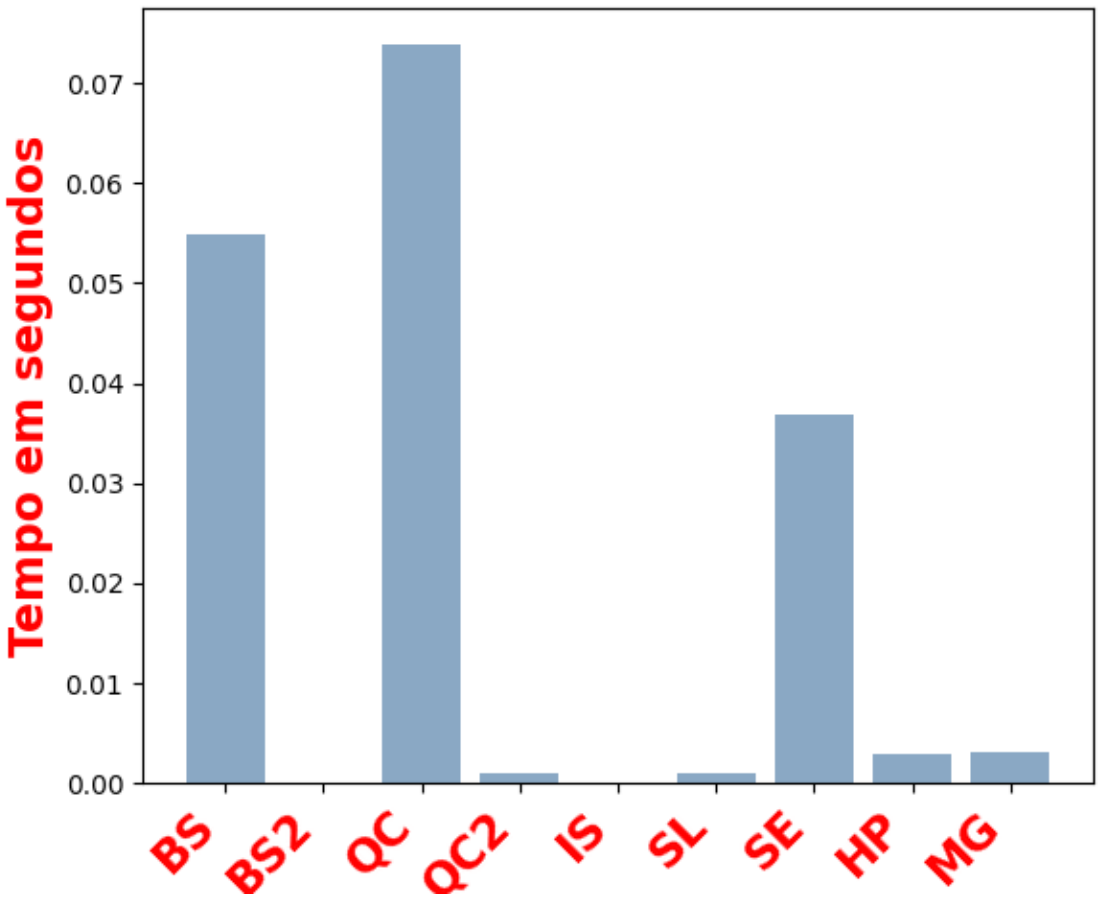
variável **clock_end** que também receberá o método **time.time()**, assim dizendo, **clock_end = time.time()**. Portanto temos dois tempos (clock_start e clock_end) que subtraídos (clock_end - clock_start) resultam no tempo de duração do algoritmo de ordenação em segundos. O cálculo de tempo é feito para cada algoritmo separadamente, portanto quando um algoritmo é finalizado, as variáveis (clock_start e clock_end) são zeradas e uma nova contagem recomeça.

Exemplo de medição dos tempos em segundos (Entrada 1000, Modo 2).

```
Algoritmo   | tempo
BubbleSort  |  0.0549163818359375  seg.
BubbleSort 2|  0.0  seg.
Quicksort   |  0.07380270957946777  seg.
Quicksort 2 |  0.0009984970092773438  seg.
Insertion   |  0.0  seg.
Shellsort   |  0.0009949207305908203  seg.
Selection   |  0.03689408302307129  seg.
Heapsort    |  0.0029952526092529297  seg.
Mergesort   |  0.003044605255126953  seg.
```

Em nosso programa foram desenvolvidos gráficos para melhor visualização local dos dados (usuário não precisará de um software externo para a visualização gráfica das informações). Com a importação da bilbioteca matplotlib.pyplot (import matplotlib.pyplot as plt) e com o uso dos devidos métodos temos:

Exemplo de plotação de gráfico local (Entrada 1000, Modo 2).

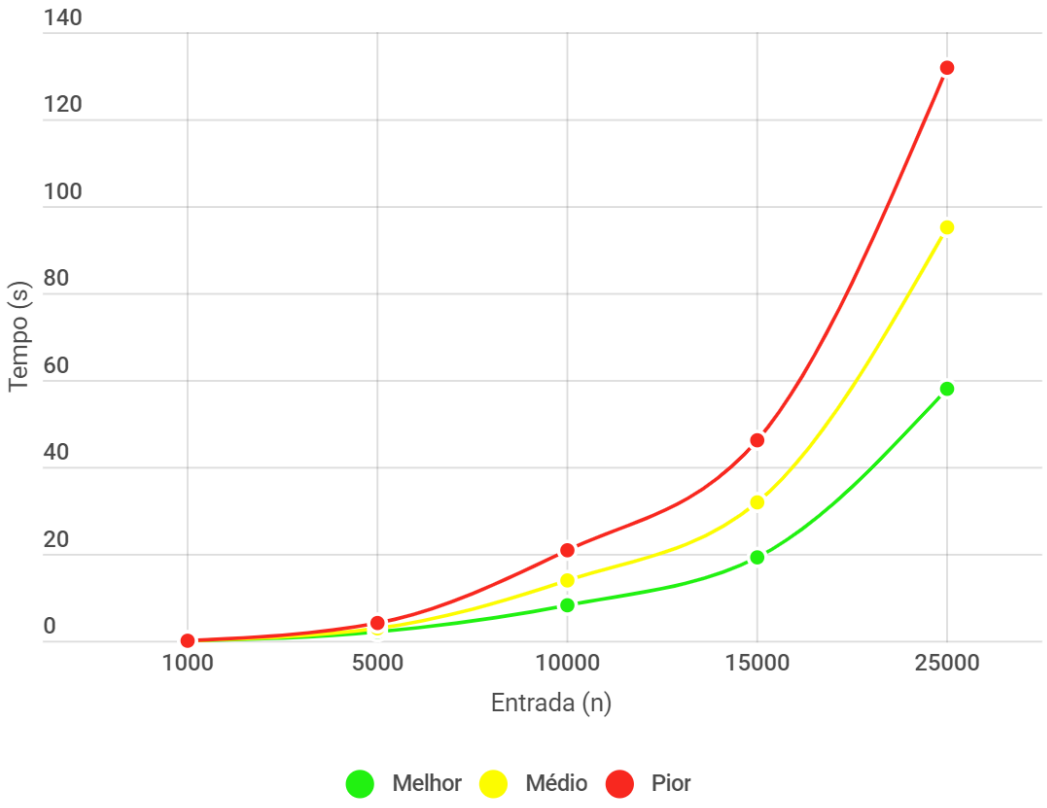


Alguns algoritmos são muito rápidos com pequenas entradas e modos específicos, como visto acima, logo para nossa análise decidimos realizar uma média de três teste para cada entrada e modo. Portanto nossos resultados derivam dessa média que busca evitar tal tempo indesejado de execução.

Resultados

BubbleSort

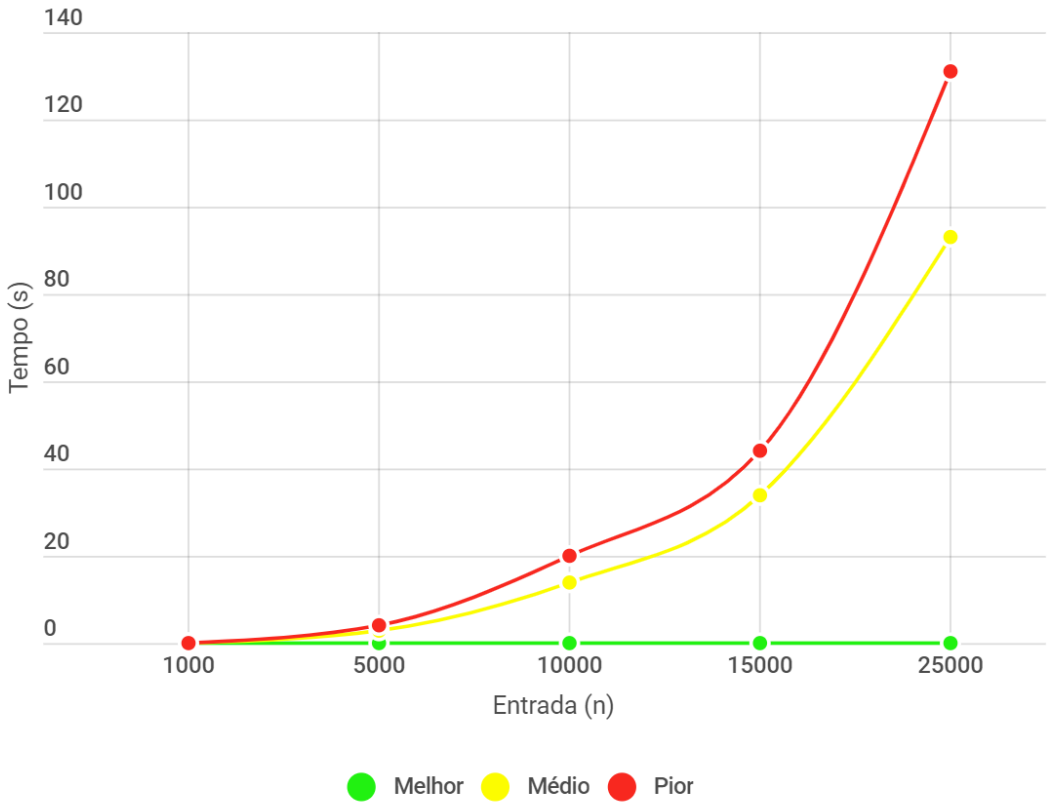
Entrada	Melhor	Médio	Pior
1000	0.0750	0.1266	0.1643
5000	2.0255	3.6387	4.8718
10000	8.7941	14.6049	21.2316
15000	19.6395	32.5675	46.8617
25000	58.3967	95.8530	132.7602



O melhor, pior e médio casos correspondem, respectivamente, ao modo crescente, decrescente e aleatório. Com o aumento do valor da entrada o tempo de execução cresce de forma proporcional, para n igual a 1000 a diferença de tempo é minima, com 5000 passa a ser basicamente de 1 segundo e ao crescimento do tamanho do vetor a diferença é clara com surpreendentes 58 segundos no melhor caso em comparação com os 132 segundos do pior para a entrada de 25000.

BubbleSort melhorado

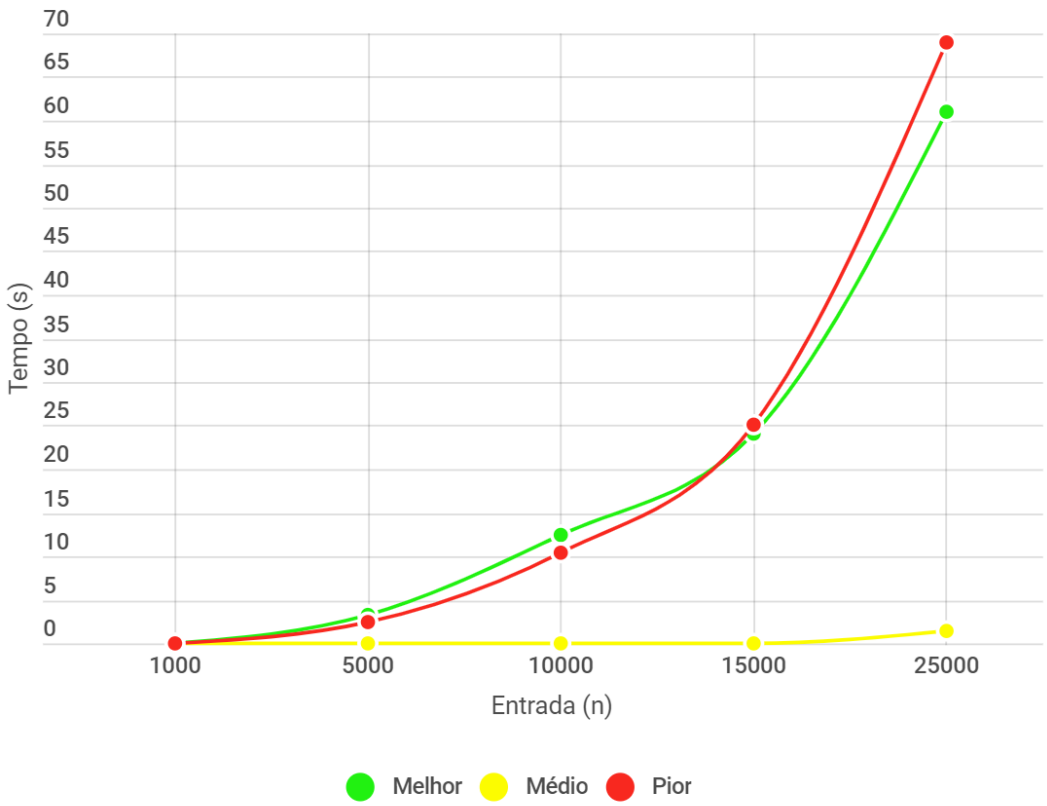
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0001	0.1427	0.1783
5000	0.0009	3.7794	4.7103
10000	0.0025	14.5737	20.0030
15000	0.0030	34.2244	44.3929
25000	0.0060	93.5731	131.7616



Com a interrupção das passagens pelo vetor quando não ocorrem mais trocas, é evitado que sejam realizados iterações desnecessárias onde o vetor é ordenado crescentemente (melhor caso), resultando em tempo de execução práticamente constante. Ambos os casos médio e pior são pouco impactados pela mudança do algoritmo. Como vimos, o BubbleSort melhorado é extremamente recomendado para pequenas entradas em comparação com o BubbleSort original. O melhor caso é da ordem de $\Theta(n)$, já o pior $\Theta(n^2)$.

QuickSort (pivô no início)

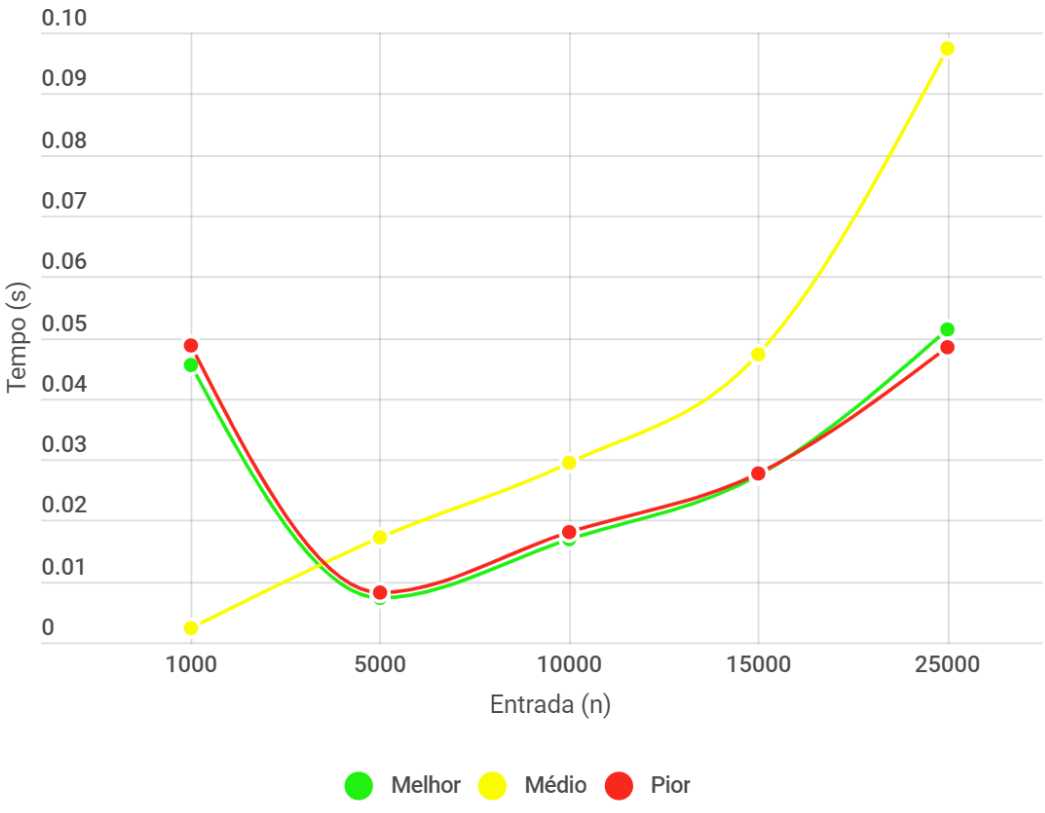
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0516	0.0023	0.0496
5000	3.1766	0.0150	2.4386
10000	12.4424	0.0503	10.4195
15000	24.0972	0.0806	25.1981
25000	61.1144	1.5250	68.9712



O caso médio (elementos dispostos de forma aleatória no vetor) teve o melhor tempo de execução em relação aos outros casos, onde se manteve constante durante todo o teste. Inicialmente o vetor ordenado decrescentemente (pior) é mais rápido em relação ao vetor ordenado crescentemente (melhor) como pode ser visto no gráfico acima, porém há uma ruptura a partir da entrada 15000, e mesmo que similares, o pior começa a crescer de forma mais intensa.

QuickSort (pivô no centro)

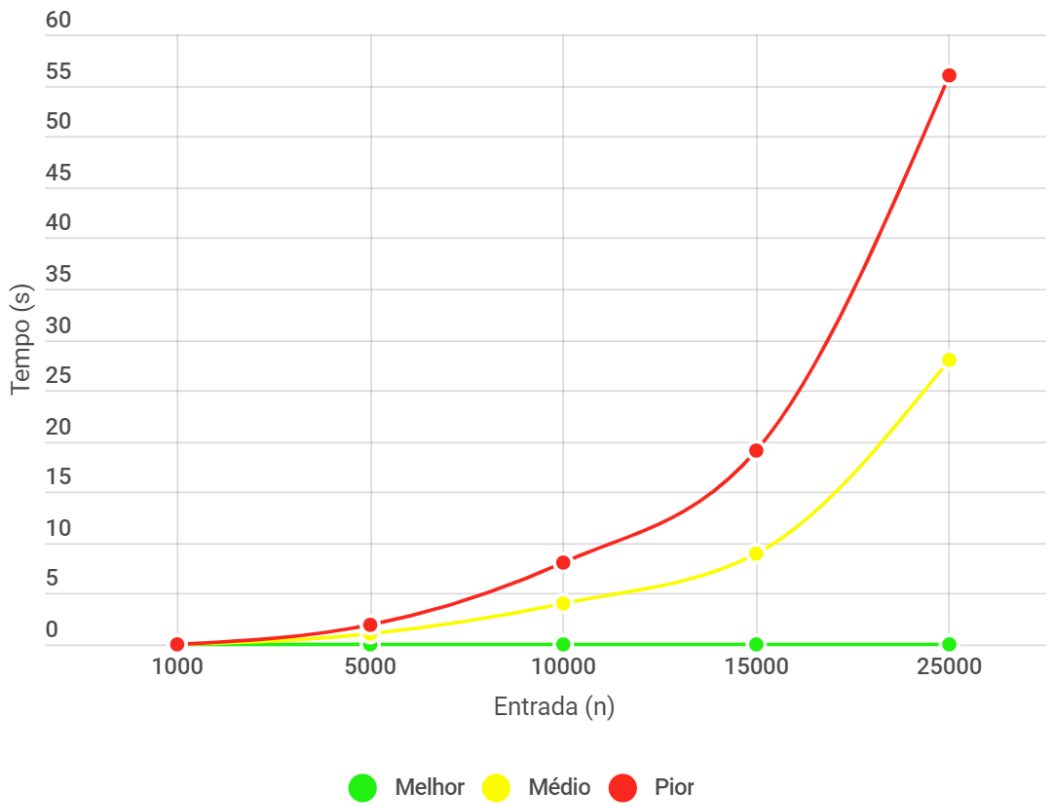
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0456	0.0023	0.0486
5000	0.0073	0.0173	0.0083
10000	0.0169	0.0295	0.0182
15000	0.0273	0.0473	0.0276
25000	0.0513	0.0973	0.0484



Muito mais rápido que o QuickSort com pivô no início, apresenta um começo similar, onde o caso médio tem tempo de execução menor que os outros casos, só que com o aumento do valor da entrada, os casos onde o vetor é ordenado (crescente e decrescente) despencam e seguem lado a lado até o final dos testes. Enquanto isto, o vetor com elementos aleatórios cresce. A diferença entre os casos não passa de 1 décimo de segundo, então para o QuickSort com pivô no centro nessas condições, pouco importa sua disposição.

InsertionSort

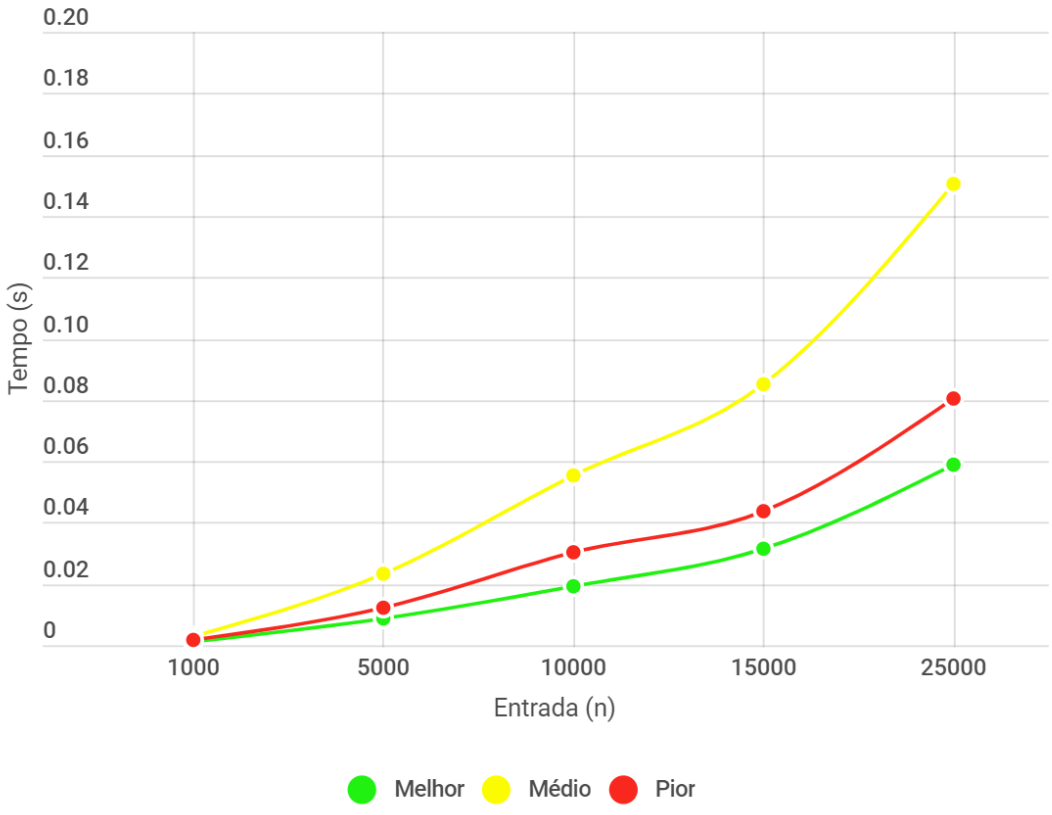
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0002	0.0436	0.0736
5000	0.0009	1.0877	2.0085
10000	0.0026	4.2750	8.5140
15000	0.0042	9.9928	19.1216
25000	0.0066	28.4527	56.9929



Lógicamente esperado, o vetor ordenado crescentemente (melhor caso) apresenta o melhor tempo de execução, práticamente constante, seguido do vetor com elementos aleatórios (caso médio) e vetor ordenado decrescentemente (pior caso). Apesar de sua complexidade $O(n^2)$ e desempenho ruim (caso médio e pior caso), ainda é uma escolha melhor se comparado a outros algoritmos de mesma complexidade, como BubbleSort.

ShellSort

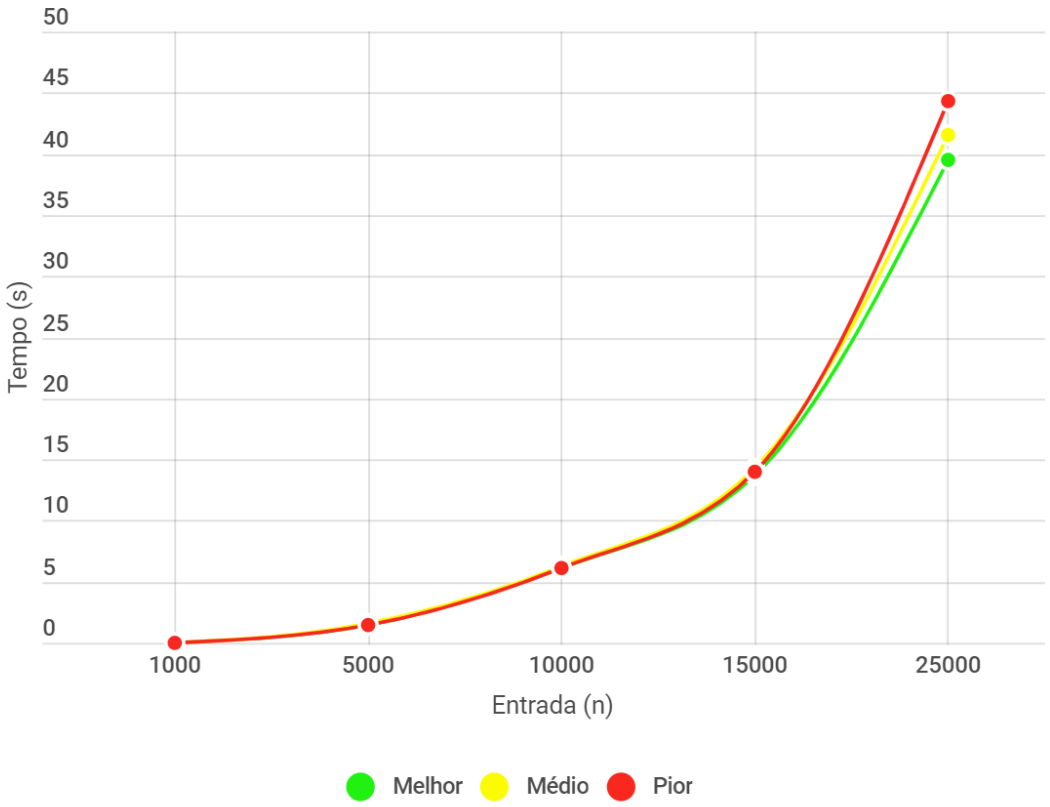
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0013	0.0028	0.0020
5000	0.0086	0.0233	0.0120
10000	0.0190	0.0553	0.0303
15000	0.0316	0.0853	0.0440
25000	0.0590	0.1506	0.0806



Assim como o QuickSort com pivô no centro, ShellSort apresentou ótimos tempos de execução. Curiosamente o vetor ordenado decrescentemente (pior caso) ordenou mais rapido que o vetor com valores aleatórios (caso médio), porém para entradas pequenas é pouco perceptível.

SelectionSort

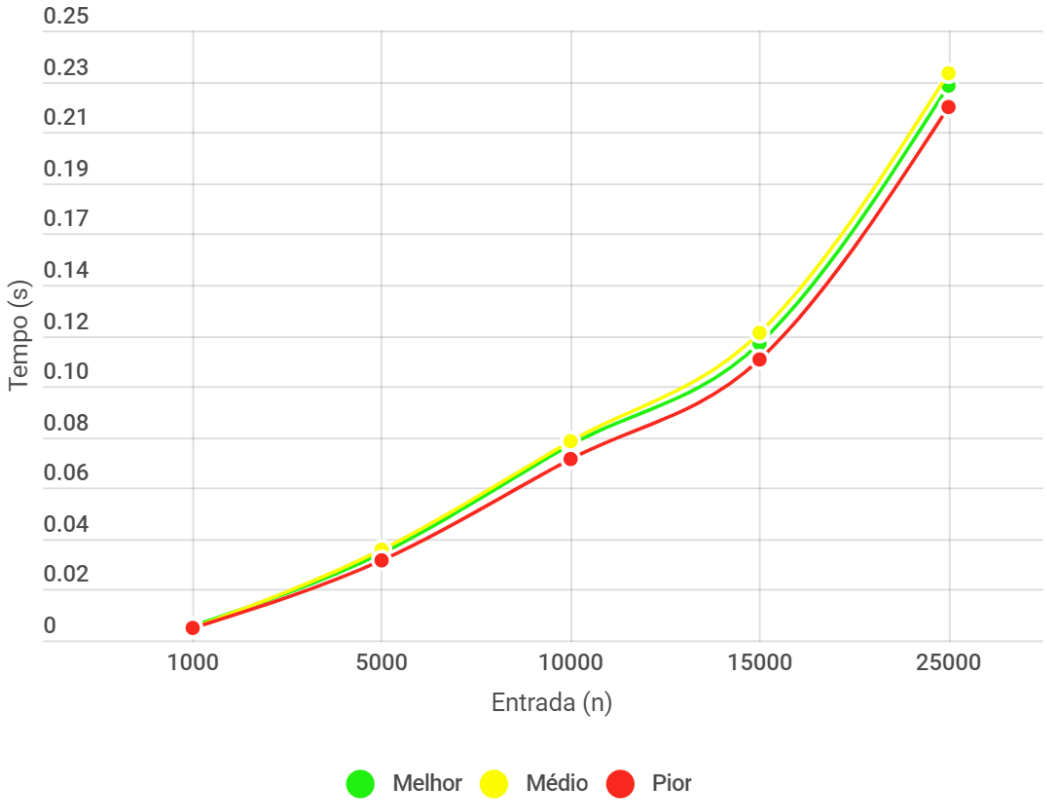
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0580	0.0576	0.0603
5000	1.4632	1.6100	1.4881
10000	6.1125	6.3225	6.1764
15000	13.7357	14.2981	13.9410
25000	39.4987	41.5021	44.2618



Analisando o algoritmo SelectionSort, percebe-se que os 3 casos são bem próximos um aos outros. De forma geral, é considerado um algoritmo de ordenação lento, e como visto nos teste acima, tal afirmação comporta-se. Todos os casos têm complexidade de O(n²), em nossa maior entrada (25000) o melhor caso destaca-se por 5 segundos em relação ao pior.

HeapSort

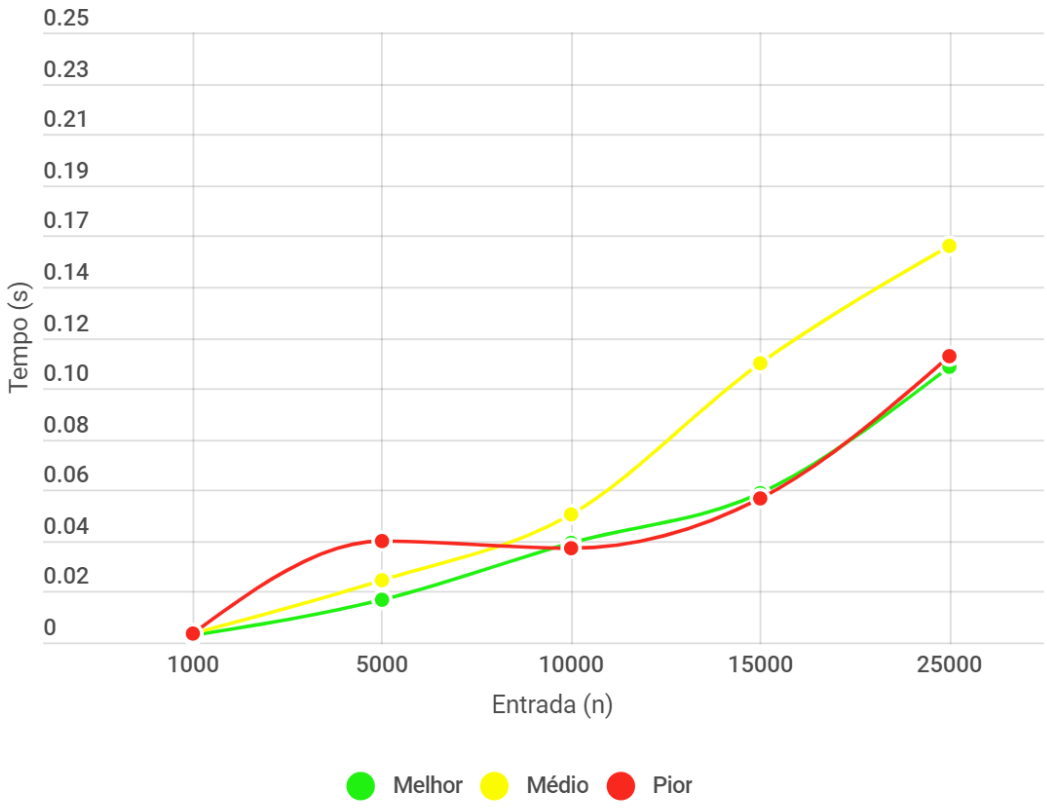
Entrada	Melhor Caso	Caso Médio	PiorCaso
1000	0.0056	0.0049	0.0050
5000	0.0356	0.0369	0.0323
10000	0.0793	0.0812	0.0738
15000	0.1205	0.1254	0.1146
25000	0.2255	0.2306	0.2173



Com complexidade O(n log n) para todos os casos, o HeapSort contém um dos melhores desempenhos e estabilidade para ordenação. Repare que a diferença de tempo é miníma comparando os casos de cada entrada, até mesmo para 25000 elementos a diferença máxima é de um centésimo de segundo.

MergeSort

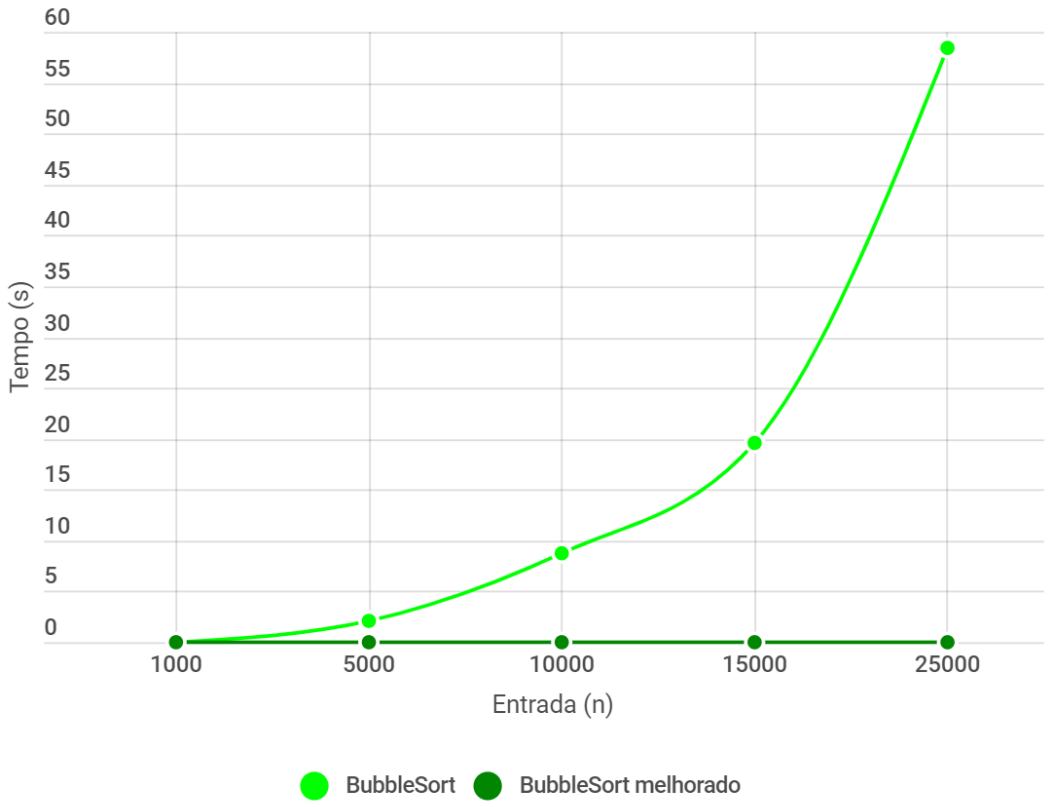
Entrada	Melhor Caso	Caso Médio	Pior Caso
1000	0.0030	0.0037	0.0033
5000	0.0175	0.0256	0.0413
10000	0.0404	0.0518	0.0381
15000	0.0606	0.1134	0.0583
25000	0.1122	0.1616	0.1163



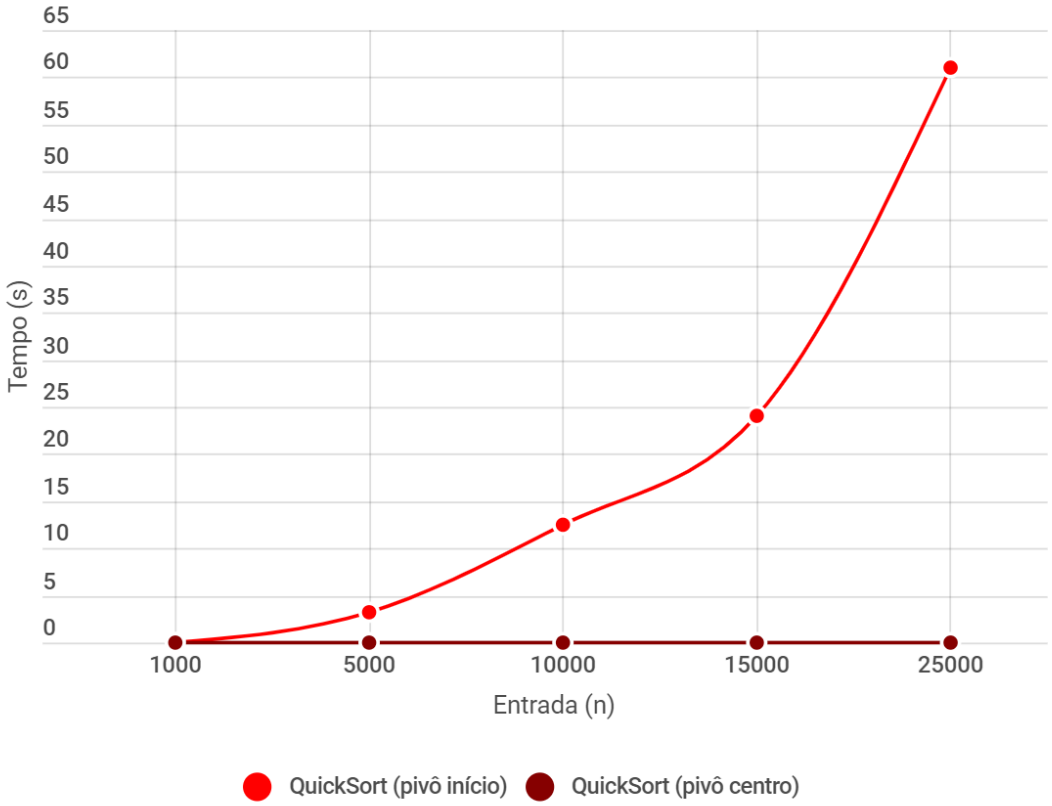
A complexidade do algoritmo MergeSort, que utiliza o método de divisão e conquista, é de O (n log n), sendo mais eficiente que os algoritmos BubbleSort, SelectionSort e InsertionSort. Até 5000 elementos os tempos seguiram respectivamente, melhor, médio e pior caso, porém a partir de 10000 o médio caso cresce, e o pior e melhor seguem similares. Assim como HeapSort, ShellSort e QuickSort (pivô no centro) o MergeSort apresentou ótimos tempos de execução para ordenação de um vetor de elementos.

Comparações entre algoritmos

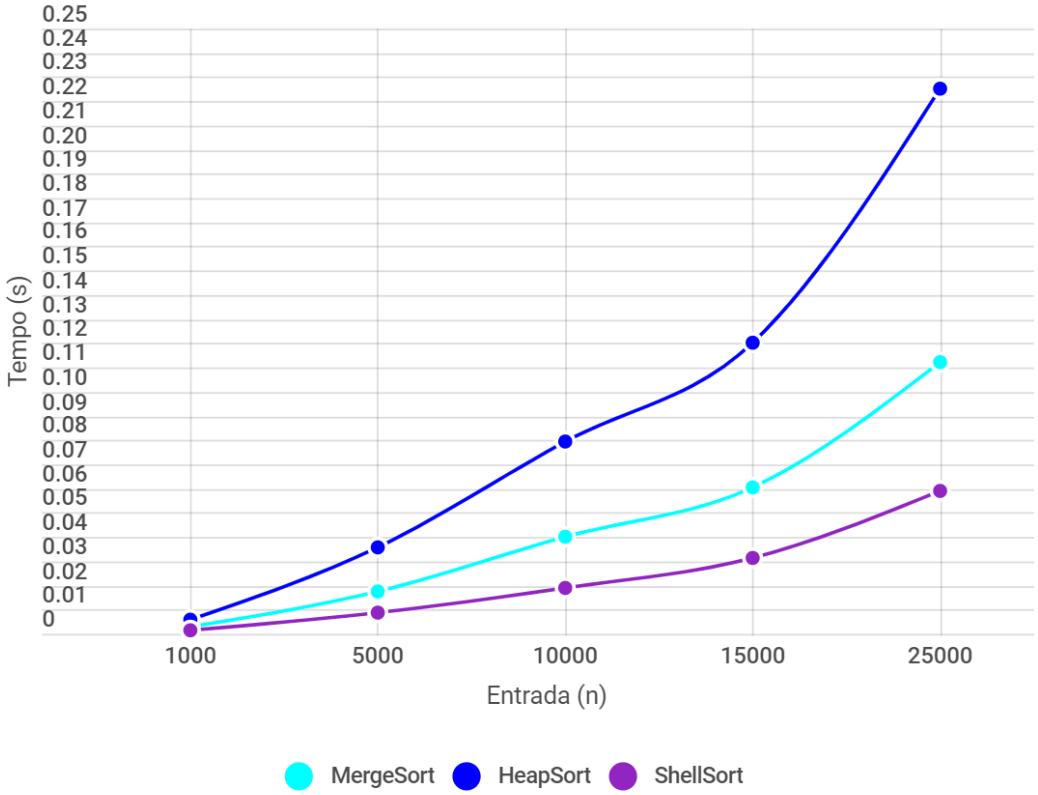
BubbleSort x BubbleSort melhorado



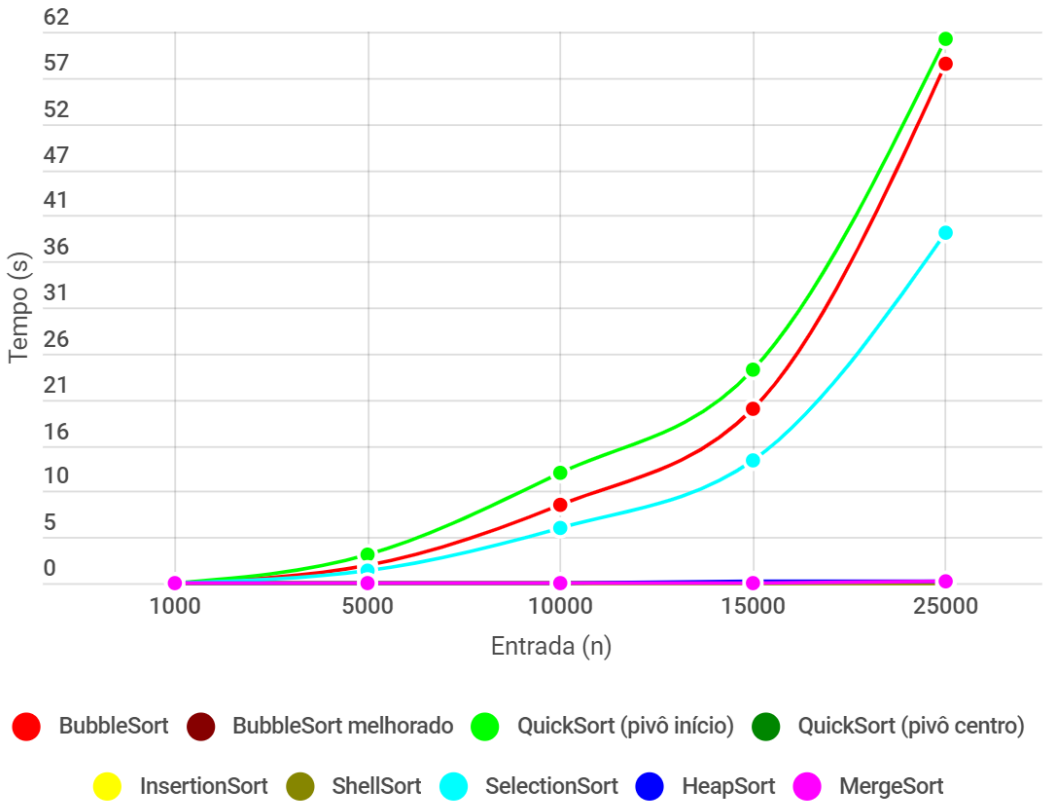
QuickSort (pivô início) x QuickSort (pivô centro)

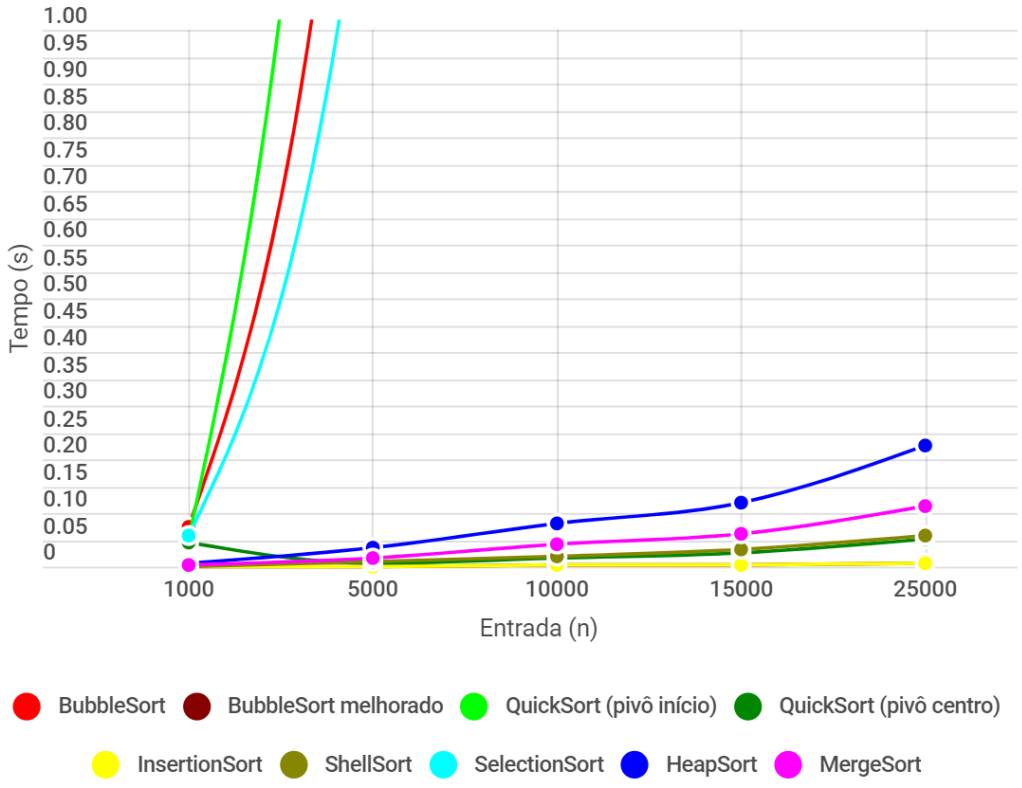


MergeSort x HeapSort x ShellSort



Todos os algoritmos





Conclusão

Com base no que foi apresentado, este relatório tem como objetivo o estudo dos algoritmos de ordenação, assim como análise experimental e assintótica. Desenvolvido em Python3, o programa análisa o tempo de execução/ordenação dos algoritmos para diferentes entradas (1000, 5000, 10000, 15000 e 25000) e modos (melhor, médio e pior caso). Observamos que os algoritmos de complexidade O(n log n) tiverem os melhores tempos de ordenação em todos os casos, porém vale ressaltar que, para vetores ordenados crescentemente (melhor caso) o método InsertionSort é uma ótima opção, resultando em um dos melhores tempos de execução. Como expectamos, os algoritmos de complexidade $O(n^2)$ manifestaram altos tempos de ordenação com aumento das entradas dos vetores e a forma que eram dispostos.
Tal implementação demonstrou de forma concreta o impacto da complexidade dos algoritmos nos tempos de ordenação dos vetores de dados.

Referências

https://daniloeler.github.io/teaching/PAA2020/index.html

https://www.geeksforgeeks.org/bubble-sort/

https://pt.wikipedia.org/wiki/Quicksort

https://www.geeksforgeeks.org/insertion-sort/

https://pt.wikipedia.org/wiki/Insertion_sort

https://www.programiz.com/dsa/shell-sort#:~:text=Shell%20sort%20is%20an%20algorithm,a%20specific%20interval%20are%20sorted.

https://www.geeksforgeeks.org/selection-sort/

https://www.faceprep.in/algorithms/selection-sort/

https://www.blogcyberini.com/2018/07/merge-sort.html

https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

https://www.inf.ufsc.br/~r.mello/ine5384/17-OrdenacaoDados3.pdf