RELATÓRIO

Comparação de Algoritmos de Ordenação

Relatório realizado por:

André Letras 50032533

Danilo Rafael 50036533

João Gonçalves 50032753

Índice

Introdução

Os algoritmos de ordenação são algoritmos utilizados para conseguir colocar elementos encontrados numa certa sequência, desordenada ou não, de forma ordenada. O objetivo é conseguir ordenar os elementos completamente no menor tempo possível.

Uma das vantagens destes algoritmos é a posterior possibilidade de se conseguir aceder aos dados ordenados de forma mais eficiente.

Este trabalho irá assim dar a conhecer os diferentes algoritmos de ordenação utilizando listas de números inteiros para representação. Os algoritmos são: Quick Sort, e dentro deste algoritmo teremos 4 variações, o QuickSort com o pivot no início, com pivot na mediana da lista, com pivot no último valor da lista e com pivot aleatório, depois implementaremos o algoritmo Merge Sort, e adicionaremos ainda, por iniciativa própria, o Bubble Sort para comparar aos algoritmos pedidos.

Os algoritmos estarão implementados na linguagem de programação Python, e faremos os vários benchmarkings para o estudo dos mesmos. Para isso iremos testar cada um dos algoritmos em listas de 200, 400, 800, 1600, 3200 e 5000 números inteiros, respetivamente, e em 4 tipos de listas: listas aleatórias, listas ordenadas ascendentes e descendentes e ainda em listas parcialmente ordenadas. Antes da implementação iremos também realizar o estudo teórico da complexidade dos vários algoritmos já mencionados.

Análise Teórica

QuickSort

Este é um algoritmo de ordenação que utiliza o paradigma "Dividir para Conquistar". Para isso utiliza funções recursivas, onde vai dividindo uma lista em sublistas mais pequenas, com objetivo de a ordenar por inteiro. Este algoritmo tem como função principal a divisão (partition()) da lista. O objetivo desta divisão é dado uma lista de input, conseguir escolher um elemento chamado pivot. Ao escolher o pivot, o algoritmo irá posicionar todos os elementos que forem menores que este à sua esquerda, e consequentemente, os elementos maiores à sua direita. De seguida aplica este método às sublistas criadas.

Existem assim várias implementações para a colocação do pivot, no entanto para este Quicksort. Nós utilizámos quatro implementações diferentes para verificar o comportamento dos algoritmos. Utilizámos o pivot como elemento inicial da lista, como elemento final da lista, como elemento mediano de uma lista e ainda utilizámos o QuickSort com pivot selecionado aleatoriamente.

Complexidade algorítmica do QuickSort (pivot – elemento inicial)

Em termos de complexidade temporal deste algoritmo, o melhor caso irá ocorrer quando as divisões forem balanceadas, ou seja, cada partição com **n/2** elementos.

Como tal, **T(n) = 2T(n/2) + O(n)** é a expressão para o melhor caso. Neste caso, a solução é de **T(n) = O(n.logn)**, ou, **O(logn).**

O pior caso irá ocorrer quando as funções recursivas produzirem divisões com 0 e n-1 elementos. Isto acontece quando o array está ordenado por ordem crescente ou decrescente.

Como tal, **T(n) = T(n-1) + O(n)** é a expressão para o pior caso possível, onde se conclui que **O(n) = O(n^2)**

Complexidade algorítmica do QuickSort (pivot – elemento final)

Esta alteração é apenas na escolha do pivot. Como tal, ao invés de se escolher como pivot o elemento inicial da lista, escolhe-se o elemento final para dividir a lista e realizar as comparações. Não tem qualquer diferença face à implementação inicial do QuickSort uma vez que o número de interações entre esta implementação e a que possui o pivot como elemento inicial é igual.

Complexidade algorítmica do QuickSort (pivot – elemento mediano)

Esta variação do QuickSort é apenas na escolha do pivot. O pivot escolhido é o elemento médio da lista. A vantagem deste algoritmo é que para listas ordenadas consegue sempre apresentar valores muito mais rápidos de ordenação, uma vez que a lista está balanceada desde a primeira iteração do algoritmo. Isso faz com que seja muito difícil alcançar a complexidade de **O(n^2),** sendo o caso geral **O(log n).**

Complexidade algorítmica do QuickSort (pivot – aleatório)

Esta implementação tem complexidade igual ao QuickSort inicial com pivot como elemento inicial da lista. No entanto, acaba por ser bem mais imprevisível, pois pode escolher um valor que seja considerado um pior caso para o algoritmo, e em vez de apresentar uma complexidade de **O(log n)** acaba por apresentar uma complexidade de **O(n^2),** sendo mais lento na sua execução.

MergeSort

Este algoritmo faz parte também da premissa de "Dividir para Conquistar", em que uma lista é primeiramente dividida ao meio e criar duas sublistas de elementos. Depois é replicado recursivamente a todas as sublistas até restar listas de apenas um elemento. Depois é feita a comparação de cada elemento e organizado por ordem para formar a lista inicial já organizada.

Complexidade do algoritmo MergeSort

Este algoritmo vai ter complexidade igual comparativamente ao estudado anteriormente – QuickSort. Uma vez que vai ter de realizar divisões, o tempo dado para a função recursiva de realizar a divisão é de **T(n) = 2T(n/2) + O(n)**. Como tal, para satisfazer esta condição, o algoritmo tem complexidade **O(n log n),** quer seja um melhor ou pior caso, uma vez que o processo será igual para qualquer lista.

BubbleSort

Ao contrário dos algoritmos apresentados anteriormente, este algoritmo é iterativo. Isto significa que, por esta razão já irá ter complexidade pior do que QuickSort e MergeSort. O funcionamento deste algoritmo, consiste na passagem por toda a lista ao passo que vai realizando as trocas dos elementos e as comparações. É bastante bom para listas ordenadas ou listas quase ordenadas, no entanto o QuickSort acaba por ser sempre superior em termos de tempo.

Complexidade do algoritmo BubbleSort

A complexidade deste algoritmo é bastante complexa uma vez que, sendo o mesmo iterativo, possui dois ciclos integrados para realizar a comparação dos elementos. Apenas irá parar quando encontrar o item número (n-2) e (n-1), uma vez que o index das listas começa no número 0.

Para isso a complexidade das comparações é dada por: (n/2)*(n-1). Temos então:

$$T(n) = 2 \times \frac{n(n-1)}{2} = n^2 - n$$

Conclui-se assim que a complexidade em notação Big-O do algoritmo BubbleSort é $O(n^2)$.

Análise Experimental

Para a nossa análise experimental (benchmarking), utilizámos 4 tipos de listas. Em cada tipo de listas foram dados os valores aleatórios entre 0 e 100, para listas com 200, 400, 800, 1600, 3200 e 5000 itens. Antes e depois de chamar a função de ordenação correspondente a cada algoritmo foi usado ainda o módulo "Time" do Python para conseguirmos registar o tempo passado entre o início e o fim da execução, recorrendo ao módulo time.perf_counter().

Listas Aleatórias

Para este benchmarking, utilizámos listas geradas aleatoriamente, não interessando a ordem de apresentação dos elementos na lista.

Os tempos obtidos podem ser observados na tabela1, abaixo:

n	QuickSort – elemento inicial	QuickSort – elemento aleatório	QuickSort – último elemento	QuickSort – Elemento mediano	BubbleSort	MergeSort
200	0,00122	0,00265	0,001	0,00085	0,00882	0,00187
400	0,00238	0,00506	0,00269	0,00268	0,04888	0,00407
800	0,00656	0,01049	0,00635	0,00346	0,20051	0,01379
1600	0,01534	0,02042	0,02318	0,02156	0,78736	0,01435
3200	0,04377	0,03909	0,02821	0,02243	2,96106	0,05181
5000	0,06988	0,05248	0,05415	0,02978	6,94267	0,0646

Tabela 1: Tempo, em segundos de cada algoritmo para listas aleatórias.

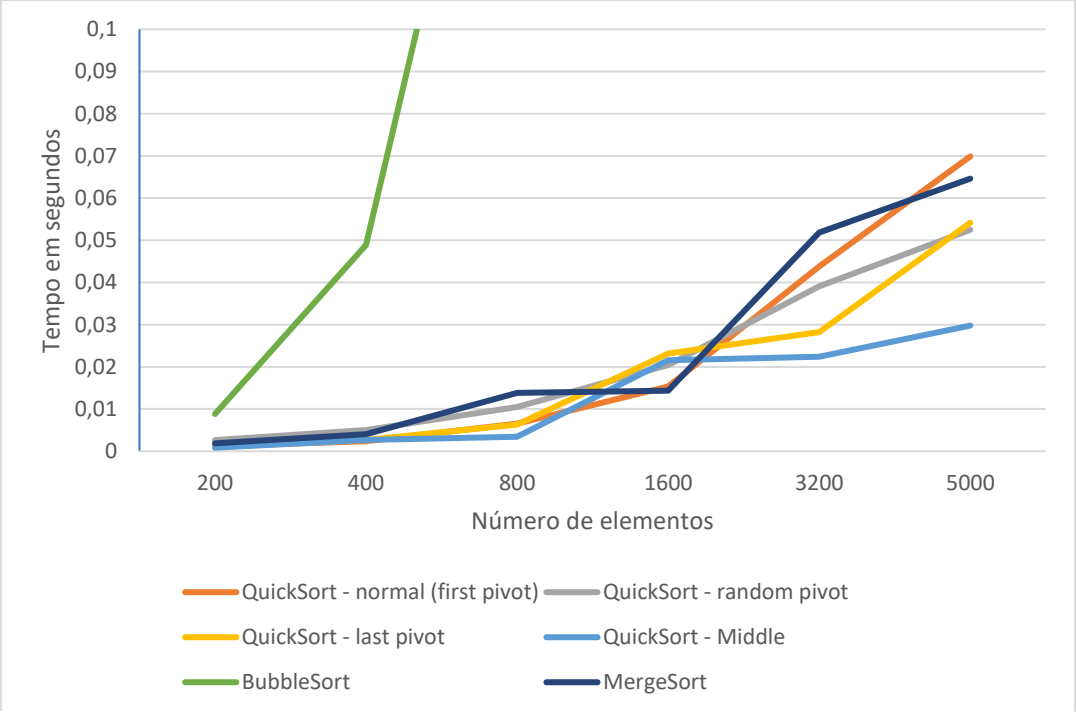

Figura 1: Gráfico dos tempos obtidos pelos algoritmos em listas aleatórias.

Na figura 1, observa-se o gráfico obtido para as listas de ordem aleatória. Conseguimos observar que todos os algoritmos recursivos (QuickSort – as 4 variantes, e o MergeSort) foram mais eficientes do que o BubbleSort que apresenta valores muito mais altos para o mesmo número de elementos.

Listas Ordenadas

Ordem Crescente

Neste caso, os algoritmos foram chamados em listas ordenadas por ordem crescente. Os tempos podem ser observados na tabela 2, abaixo:

n	QuickSort – elemento inicial	QuickSort – elemento aleatório	QuickSort – último elemento	QuickSort – Elemento mediano	BubbleSort	MergeSort
200	0,00342	0,00123	0,00389	0,00061	0,00388	0,00103
400	0,01004	0,00315	0,01355	0,00135	0,02433	0,00609
800	0,0178	0,00654	0,0265	0,0036	0,10618	0,01464
1600	0,04055	0,01479	0,05279	0,00679	0,45993	0,01488
3200	0,06771	0,03423	0,1298	0,01536	1,71347	0,03262
5000	0,13255	0,06491	0,22571	0,01962	3,90004	0,0692

Tabela 2: Tempos obtidos, em segundos, para os algoritmos corridos em listas ordenadas por ordem ascendente.

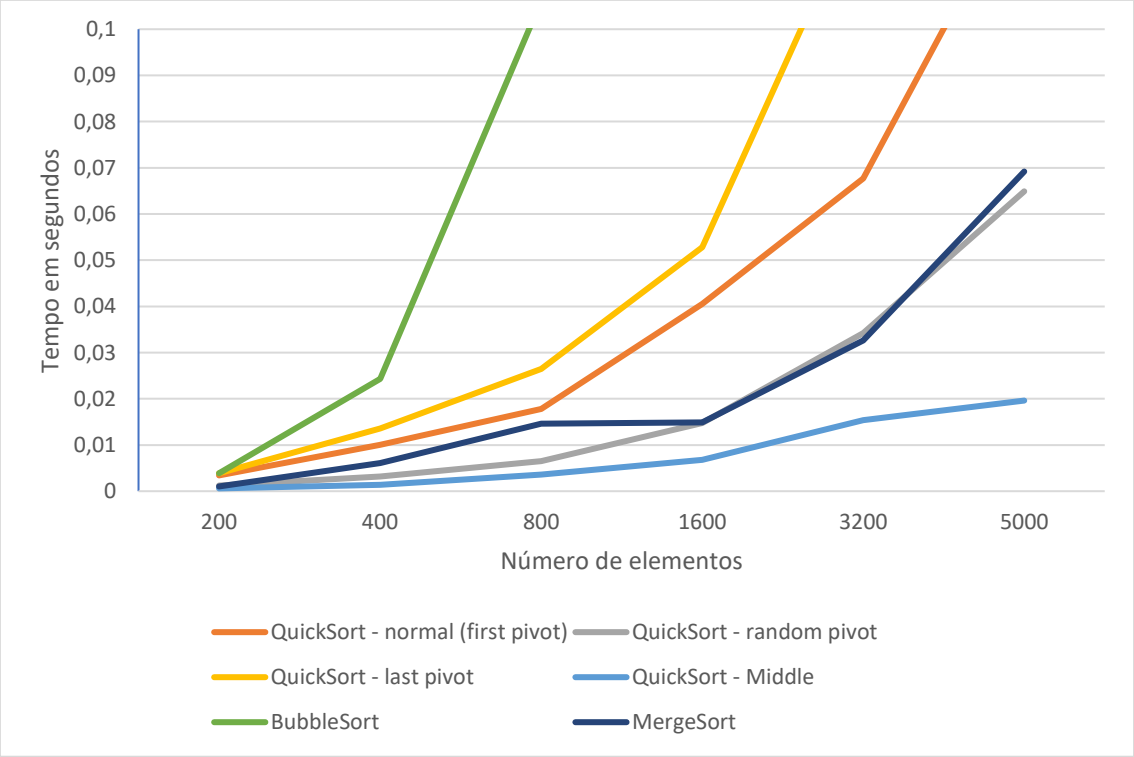

Figura 2: Gráfico dos tempos em relação ao número de elementos, dos algoritmos em listas ordenadas por ordem crescente.

No gráfico apresentado, consegue-se observar que os algoritmos QuickSort em que o pivot é considerado como primeiro ou último elemento, tiveram um comportamento semelhante, mas que se aproximaram da complexidade do pior caso: $O(n^2)$. No entanto, os restantes algoritmos, em exceção do BubbleSort tiveram o comportamento esperado apresentando valores semelhantes.

De destacar ainda o comportamento ótimo do algoritmo QuickSort com o elemento pivot na mediana da lista. Uma vez que a lista fica balanceada com esse elemento pivot é muito mais rápido realizar a verificação dos restantes elementos e demora assim muito menos tempo a concluir a ordenação.

Ordem Decrescente

Para este caso, foi apenas necessário inverter as listas e chamar o algoritmo correspondente. De realçar que os tempos relativamente ao QuickSort com pivot no elemento inicial tem três tempos em falta devido a um erro no algoritmo que não foi possível resolver. No entanto em nada afeta o estudo dos mesmos uma vez que com os 3 valores obtidos é já possível fazer a comparação com os restantes. Os tempos estão apresentados na tabela 3, abaixo:

n	**QuickSort – elemento inicial**	**QuickSort – elemento aleatório**	**QuickSort – último elemento**	**QuickSort – Elemento mediano**	**BubbleSort**	**MergeSort**
200	0,0102	0,00105	0,0078	0,00033	0,01305	0,00102
400	0,02596	0,00619	0,01026	0,00093	0,06	0,00319
800	0,11455	0,01297	0,02826	0,00205	0,28602	0,00935
1600		0,01337	0,10176	0,007	1,04631	0,01353
3200		0,03846	0,16405	0,02202	4,10596	0,03258
5000		0,06092	0,2699	0,02451	10,87931	0,0714

Tabela 3: Tempos em segundos, dos algoritmos para as listas ordenadas por ordem descendente.

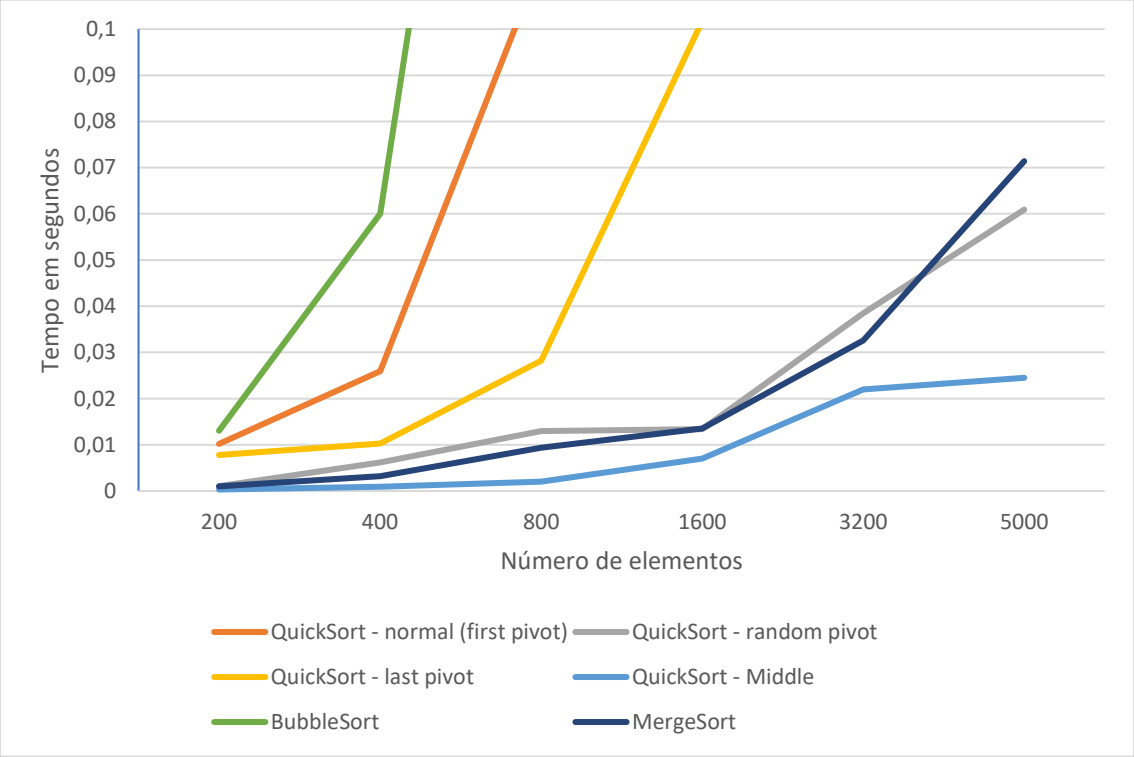

Figura 3: Gráfico dos tempos dos algoritmos de ordenação para listas ordenadas de forma descendente.

No gráfico 3, acima representado, observa-se uma subida de forma acentuada nos tempos do algoritmo QuickSort, com elementos pivot no início e no fim da lista. Isto deve-se ao facto de este representar o pior caso para este algoritmo, uma vez que este tem de realizar a verificação de toda a lista e ainda trocar todos os elementos da mesma. Como tal, observa-se que o QuickSort com elemento médio domina no tempo de execução pela mesma razão que nas listas ordenadas de forma ascendente.

O algoritmo BubbleSort continua a ser o pior em tempos, obtendo sempre a forma de gráficos com $O(n^2)$.

Listas Quase Ordenadas

Para implementar estas listas utilizámos uma função nearSorted(). Esta função faz com que a primeira parte da lista esteja por ordem, de forma crescente, mas faz a troca dos últimos elementos com os elementos médios da lista.

Os tempos para estas listas estão apresentados na tabela 4, abaixo:

n	QuickSort – elemento inicial	QuickSort – elemento aleatório	QuickSort – último elemento	QuickSort – Elemento mediano	BubbleSort	MergeSort
200	0,00621	0,00116	0,00198	0,00041	0,0062	0,00276
400	0,00988	0,00349	0,00913	0,00319	0,05073	0,00621
800	0,03412	0,01254	0,01997	0,0029	0,15969	0,01522
1600	0,15313	0,01947	0,03234	0,00475	0,59781	0,02787
3200		0,03868	0,09621	0,01329	2,37578	0,03747
5000		0,05612	0,13679	0,01985	5,40114	0,04474

Tabela 4: Gráfico dos tempos dos algoritmos de ordenação para listas quase ordenadas.

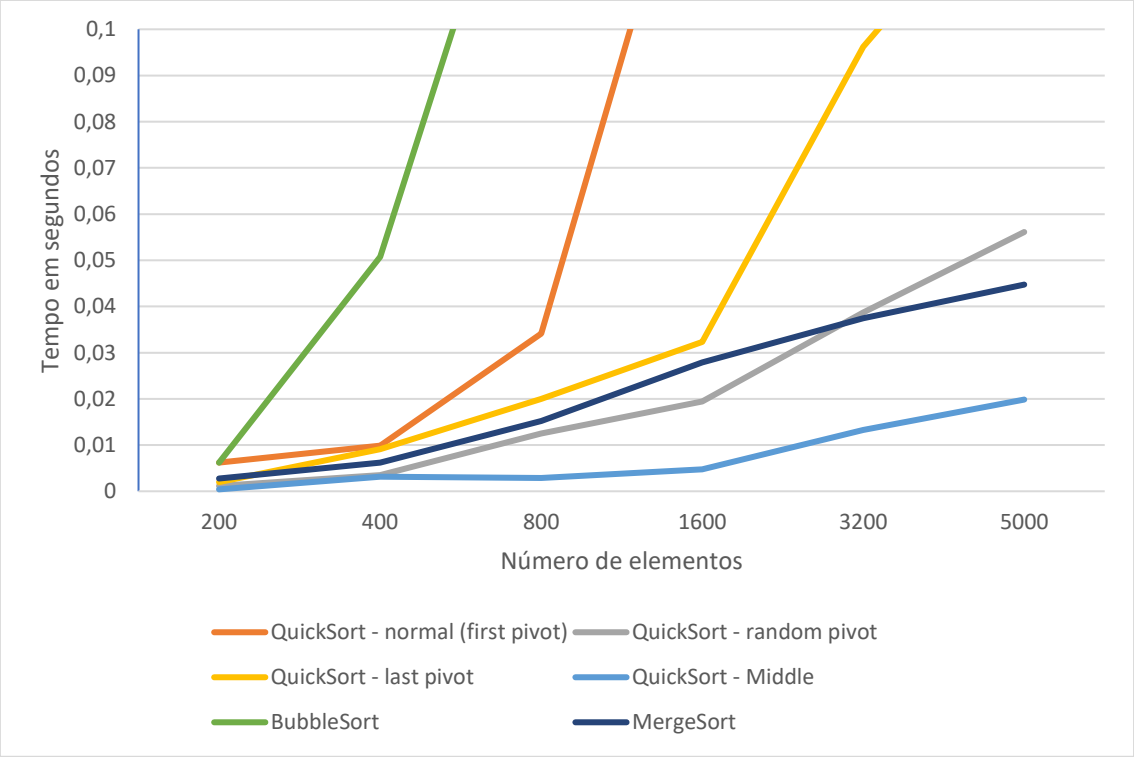

Figura 4: Gráfico dos tempos dos algoritmos de ordenação para listas quase ordenadas.

No gráfico acima, figura 4, consegue-se observar os tempos obtidos para as listas quase ordenadas. Realça-se que aconteceu o mesmo em relação ao primeiro algoritmo, que apresenta erro quando implementados muitos elementos na lista. No entanto, fazendo a comparação, consegue-se observar o achatamento de todos os gráficos, uma vez que necessitam de menos iterações para organizar toda a lista.

O algoritmo BubbleSort consegue ser mais eficiente neste tipo de listas, no entanto, os restantes algoritmos continuam a apresentar valores mais baixos em termos de execução. Realça-se o facto de que o algoritmo QuickSort com o elemento médio como pivot é o algoritmo que melhores tempos obteve, seja para este caso seja para as restantes listas de elementos. Isso evita o facto de ser possível encontrar um pior caso na implementação da lista, e acaba por ter complexidade constante de O(n log n).

Conclusão

Em suma, consegue-se concluir que os algoritmos de ordenação recursivos apresentam, no geral tempos melhores do que algoritmos de ordenação iterativos, como o caso do BubbleSort.

O QuickSort é assim um algoritmo bastante utilizado e útil em qualquer tipo de lista, no entanto, de forma a conseguir obter menos probabilidade de ter piores casos em termos de complexidade, a forma ideal é utilizando o QuickSort com elemento pivot no index médio da lista. Observou-se que todos os valores de tempo para este algoritmo foram substancialmente mais reduzidos que no geral dos restantes algoritmos.

O algoritmo QuickSort com elemento pivot selecionado de forma aleatória conseguiu ter bom desempenho no geral, no entanto há sempre uma probabilidade de, em alguma lista, obter valores indesejados e tender para complexidade $O(n^2)$. Para isso, a melhor forma e equiparável é utilizar o algoritmo MergeSort que obteve valores extremamente idênticos em qualquer forma de lista apresentada. Desta forma, consegue-se prever sempre o resultado em termos de complexidade, uma vez que o MergeSort tem complexidade constante de O(n log n).

Referências

Miller, B. N., & Ranum, D. L. (2011). *Problem solving with algorithms and data structures using python Second Edition*. Franklin, Beedle & Associates Inc..

https://helloacm.com/quick-demonstration-of-quick-sort-in-python/

https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

https://visualgo.net/pt

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

https://www.blogcyberini.com/2018/08/quicksort-analise-e-implementacoes.html

https://www.blogcyberini.com/2018/08/quicksort-mediana-de-tres.html

https://www.blogcyberini.com/2018/09/quicksort-com-pivo-aleatorio.html

https://www.geeksforgeeks.org/3-way-quicksort-dutch-national-flag/

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

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

https://www.toptal.com/developers/sorting-algorithms