CENTRO UNIVERSITÁRIO RITTER DOS REIS
CAMPUS ZONA SUL
ANÁLISE DE ALGORITMOS

ALISSON LÜDTKE, FILIPE TIETBÖHL, GABRIEL SCHRAMM E WILLIAN CAVALHEIRO

Trabalho N2 : Análise de algoritmos de ordenação

Porto Alegre/RS

14 de Junho de 2023

Sumário

Introdução

A eficiência dos algoritmos de ordenação é fundamental na hora de selecionar um algoritmo para o desenvolvimento de sistemas e neste trabalho abordamos e analisamos de maneira comparativa seis algoritmos clássicos de ordenação com duas complexidades diferentes: Algoritmos de ordenação simples (de ordem $O(n^2)$ e Algoritmos de ordenação complexos (complexidade de tempo de $O(n \log n)$). Para os algoritmos de complexidade simples analisamos o BubbleSort, o SelectionSort e o InsertionSort e para os algoritmos de ordem complexa analisamos o MergeSort, o HeapSort e o QuickSort.

Implementamos os seis algoritmos em suas versões clássicas e com variações com melhorias utilizando o Python, criamos uma função principal para criar o vetor e ordená-lo e uma função secundária para medir o tempo de execução do algoritmo somente. Com estas medições de tempo realizamos uma análise de função de custo temporal. Fizemos as análises com vetores com diferentes quantidades de elementos de valores inteiros.

Implementação

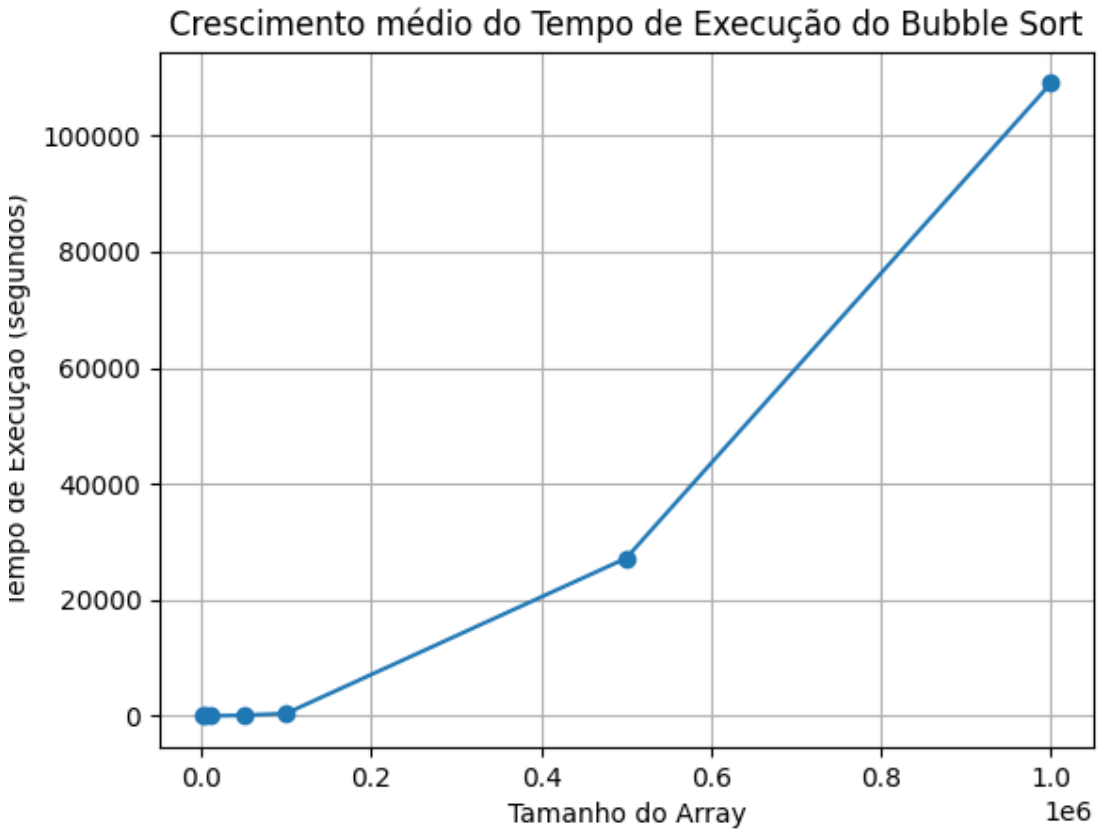
Bubble Sort

O Bubble Sort é um algoritmo que compara elementos adjacentes e os troca de lugar se estiverem na ordem errada. O algoritmo percorre repetidamente o array até que nenhum elemento precise ser trocado. A cada iteração, o elemento mais pesado (maior) "sobe" para o final do array, assim como uma bolha de ar sobe na água.

C1 -> Atribuição (n=len(arr)) = C
C2 -> Operação aritmética (n-1) = C(n+1)
C3 -> Operação aritmética (n - 1 - i) = C(n+1)
C4 -> Avaliação de expressão booleana (arr[] > arr[]) = CN
C5 -> Operação aritmética (j+1) = CN

C1+C2(N+1)+C3(N+1)+C4(N)+C5(N)

3C+4CN

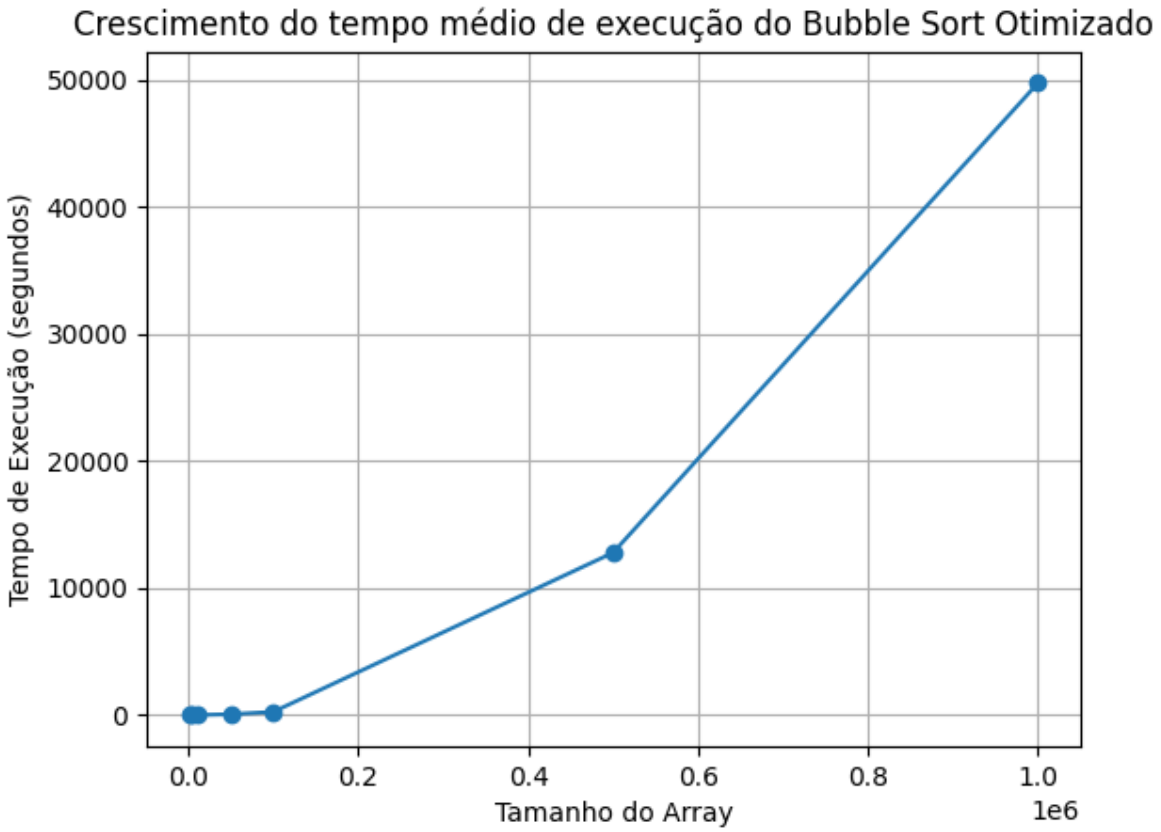


Bubble Sort Otimizado

O Bubble Sort é ineficiente quando utilizado em grandes arrays, por conta de sua complexidade de tempo ser quadrática (O(n^2)). Portanto, foi desenvolvida uma forma de deixá-lo mais eficiente.
A melhoria utilizada é adicionar uma verificação se houve alguma troca na iteração anterior. Caso não tivesse ocorrido, significa que o array já estaria ordenado e não teria necessidade de organizar.

C1 -> Atribuição (n = len(arr)) = C
C2 -> Atribuição (swapped = True) = C
C3 -> Atribuição (swapped = false) = CN+1
C4 -> Operação aritmética (i-1) = CN
C5 -> Avaliação de expressão booleana (arr[] > arr[]) = CN+1
C6 -> Atribuição (swapped = True) = CN
C7 -> Atribuição (n-=1) = CN

C1+C2+C3 (N+1)+C4 (N) + C5 (N+1)+C6(N)+C7(N)
4C+5CN

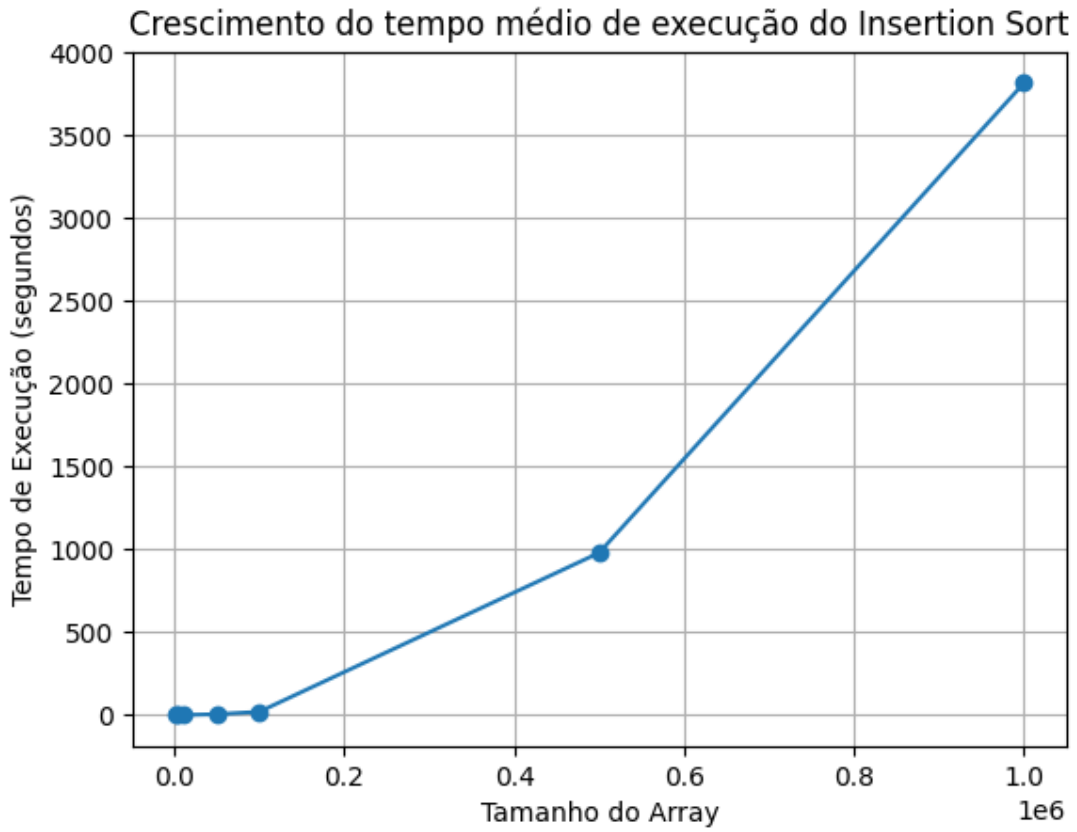


Insertion Sort

O Insertion Sort funciona dividindo o array em duas partes, a parte ordenada e a desordenada. Na fatia desordenada, o Insertion Sort encontra o menor número e o insere na posição correta, na parte ordenada, deslocando os elementos maiores para o outro lado.

C1 -> Atribuição (key = arr[i]) = CN
C2 -> Atribuição (j = i-1) = C
C3 -> Operação aritmética (i-1) = C
C4 -> Avaliação de expressão booleana (j >= 0) = CN
C5 -> Avaliação de expressão booleana (arr[j] > key) = CN
C6 -> Operação aritmética (j+1) = C
C7 -> Atribuição (arr[] = arr[]) = C
C8 -> Operação aritmética (j-=1) = CN+1
C9 -> Operação aritmética (j+1) = C
C10 -> Atribuição (arr[]=key) = C
C11 -> Retorno (return arr) = C

C1(N)+C2+C3+C4(N)+C5(N)+C6+C7+C8(N+1)+C9+C10+C11
8C+4CN



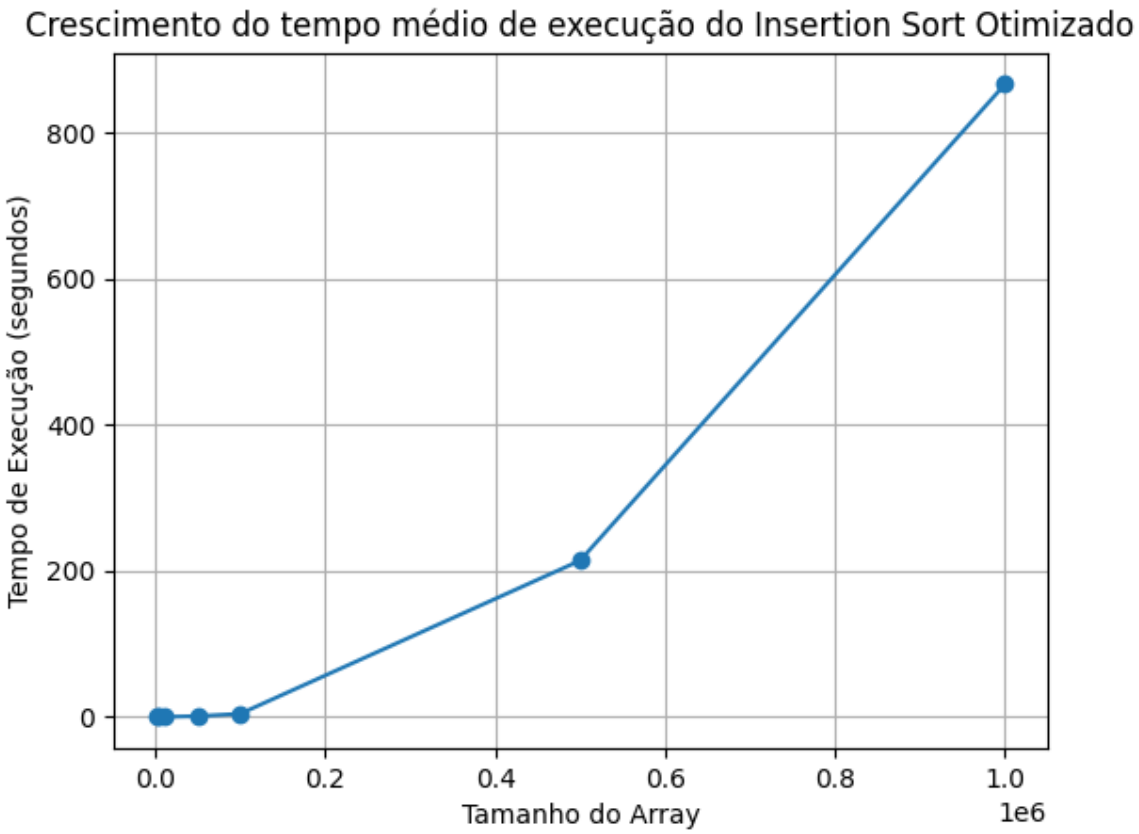
Insertion Sort Otimizado

O Insertion Sort é relativamente eficiente em conjuntos de dados pequenos, mas peca em grandes quantidades de dados, devido à sua complexidade de tempo ser quadrática.

A melhoria implementada foi realizando a troca de elementos, ao invés do deslocamento, como é normalmente realizada. Essa alteração reduz a quantidade de movimentos necessários para ordenar os elementos.

C1 -> Atribuição (key = arr[i]) = C
C2 -> Avaliação de expressão booleana (arr[j] >= key) = CN
C3 -> Atribuição (arr[] = arr[]) = C
C4 -> Retorno (arr) = C

C1+C2(N)+C3+C4
3C + CN



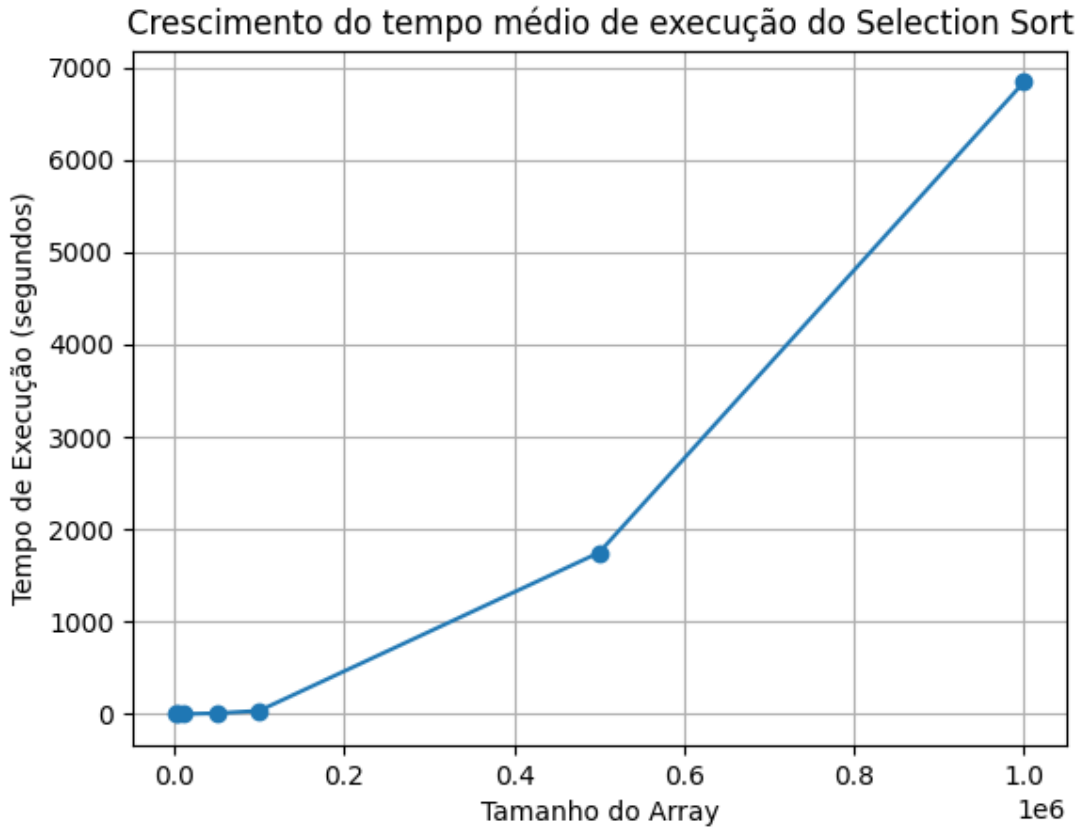
Selection Sort

O selection sort é um algoritmo de ordenação simples. Ele funciona encontrando diversas vezes o elemento de menor valor na lista não classificada e trocando ele com o primeiro elemento não classificado. Esse procedimento é feito até que a lista esteja organizada.

O algoritmo começa dividindo a lista em duas partes, uma sub-lista classificada e uma sub-lista não classificada. No começo os elementos vão todos para a sub-lista não classificada e, conforme o algoritmo é executado, ele move os elementos para a posição correta na sub-lista classificada. Dessa forma, o menor elemento é colocado na posição correta na sub-lista classificada e faz isso com todos os elementos, até que a lista fique ordenada.

C1 -> Atribuição (min_idx = i) = CN
C2 -> Operação aritmética (i+1) = CN
C3 -> Avaliação de expressão booleana (arr[] < arr[]) = CN+1
C4 -> Atribuição (min_idx = j) = CN
C5 -> Atribuição (arr[] = arr[]) = C
C6 -> Retorno (return arr) = C

C1(N)+C2(N)+C3(N+1)+C4(N)+C5+C6
3C+4CN



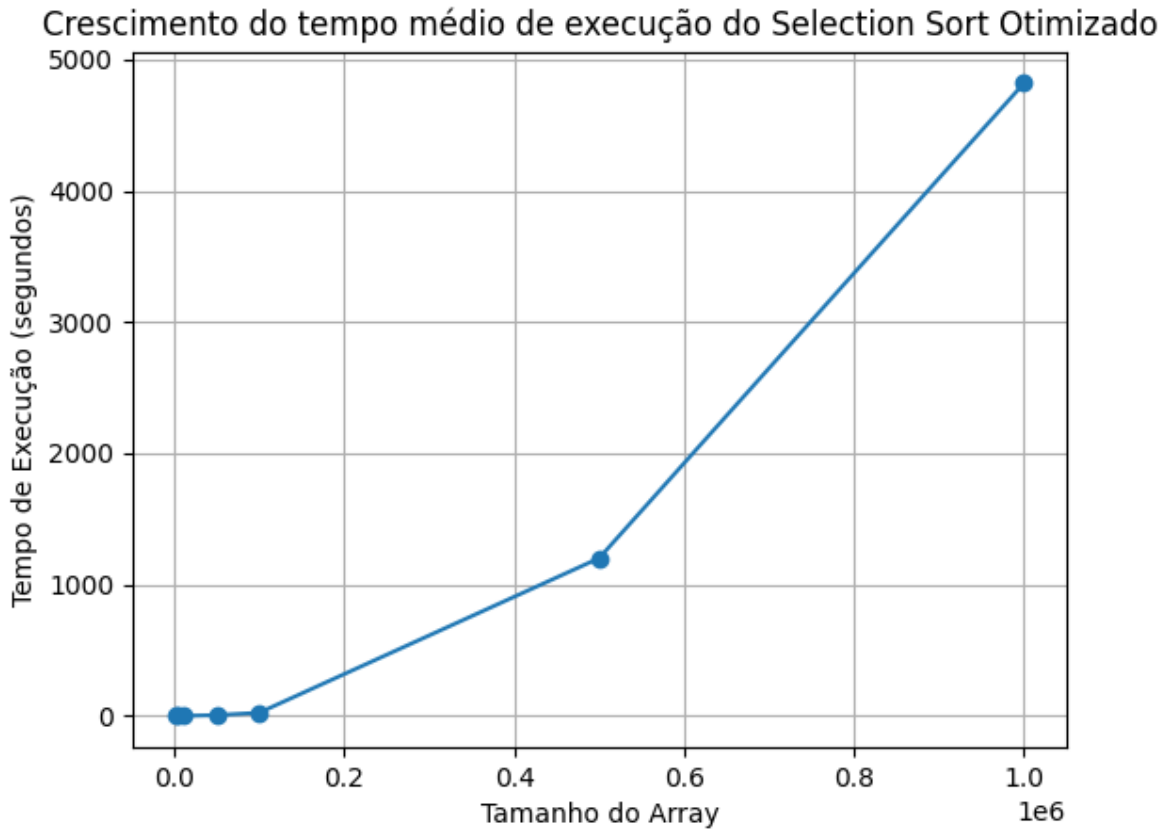
Selection Sort Otimizado

A versão otimizada do Selection Sort visa reduzir o número de trocas realizadas durante a classificação das sub-listas. Assim como sua versão original, o algoritmo começa dividindo a lista em duas sub-listas e organizando-as.
O algoritmo funciona percorrendo toda a lista não classificada primeiro e depois organizando na lista classificada de menor para maior.

C1 -> Atribuição (min_idx = i) = CN
C2 -> Operação aritmética (i+1) = CN
C3 -> Avaliação de expressão booleana (arr[] < arr[]) = CN+1
C4 -> Atribuição (min_idx = j) = CN
C5 -> Avaliação de expressão booleana (min_idx != i) = CN+1
C6 -> Atribuição (arr[] = arr[]) = CN
C7 -> Retorno (return arr) = C

C1(N)+C2(N)+C3(N+1)+C4(N)+C5(N+1)+C6(N)+C7

3C+6CN



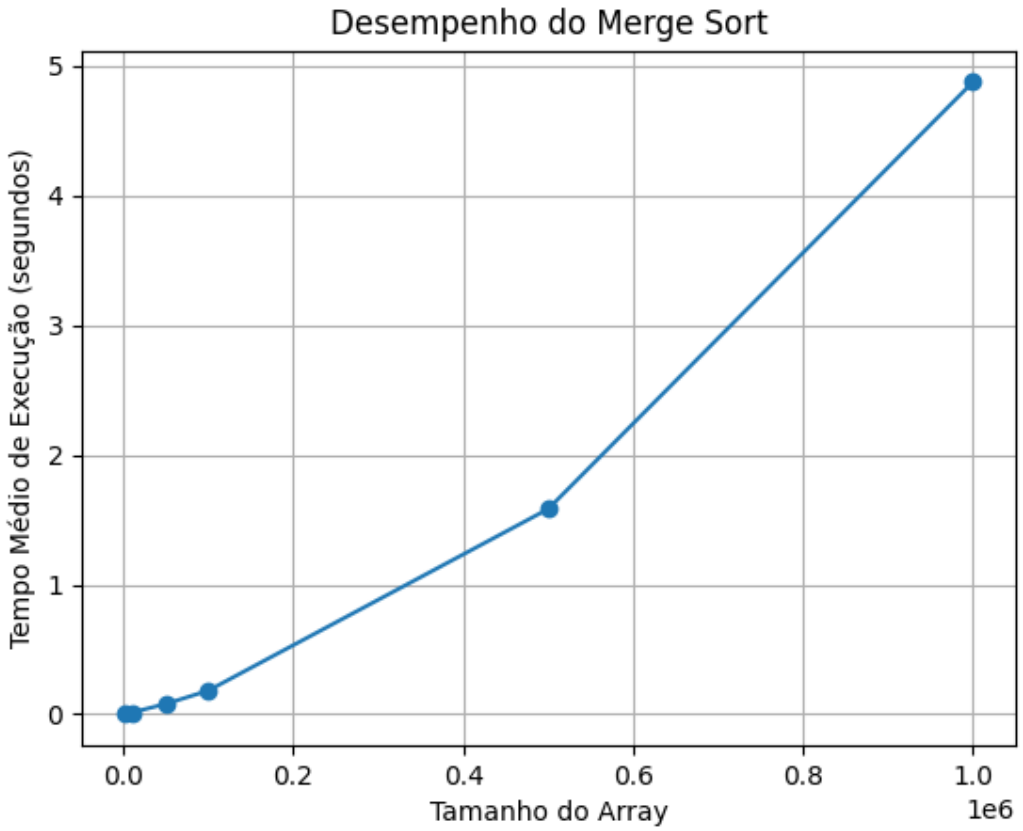
Merge Sort

O Merge Sort é o algoritmo de ordenação que funciona dividindo o array em subarrays menores e os ordenando individualmente e depois combinando os arrays ordenados para formar o array inteiro ordenado. É um algoritmo recursivo que divide o array no meio até ter somente um elemento, ele tem uma complexidade de O(n log(n)) e é usado para ordenar grandes grupos de dados e para ordenações personalizadas pois é possível ordenar dados que estão parcialmente ordenados, não ordenados e quase ordenados.

C1 -> Atribuição (início = 0) = C
C2 -> Atribuição (fim=None) = C
C3 -> Avaliação de expressão booleana (fim is none) = C(N)
C4 -> Atribuição (fim = len(array)) = C
C5 -> Operação aritmética (fim - inicio) = C(N)
C6 -> Avaliação de expressão booleana (início > 1) = C(N+1)
C7 -> Atribuição (meio = (fim + inicio) // 2) = C
C8 -> Operação aritmética (fim + inicio) = C(N)
C9 -> Operação aritmética ((fim + inicio) //2) = C(N)
C10 -> Atribuição (left = array[]) = C
C11 -> Atribuição (right = array[]) = C
C12 -> Atribuição (top_right = 0) = C
C13 -> Avaliação de expressão booleana (top_left >= len(left)) = C(N+1)
C14 -> Atribuição (array[k] = right[top_right]) = C(N+1)
C15 -> Atribuição (top_right += 1) = C (N+1)

C1+C2+C3(N)+C4+C5(N)+C6(N+1)+C7+C8(N)+C9(N)+C10+C11+C12+C13(N+1)+C14(N+1)+C15(N+1)

11C+8CN



Merge Sort Híbrido

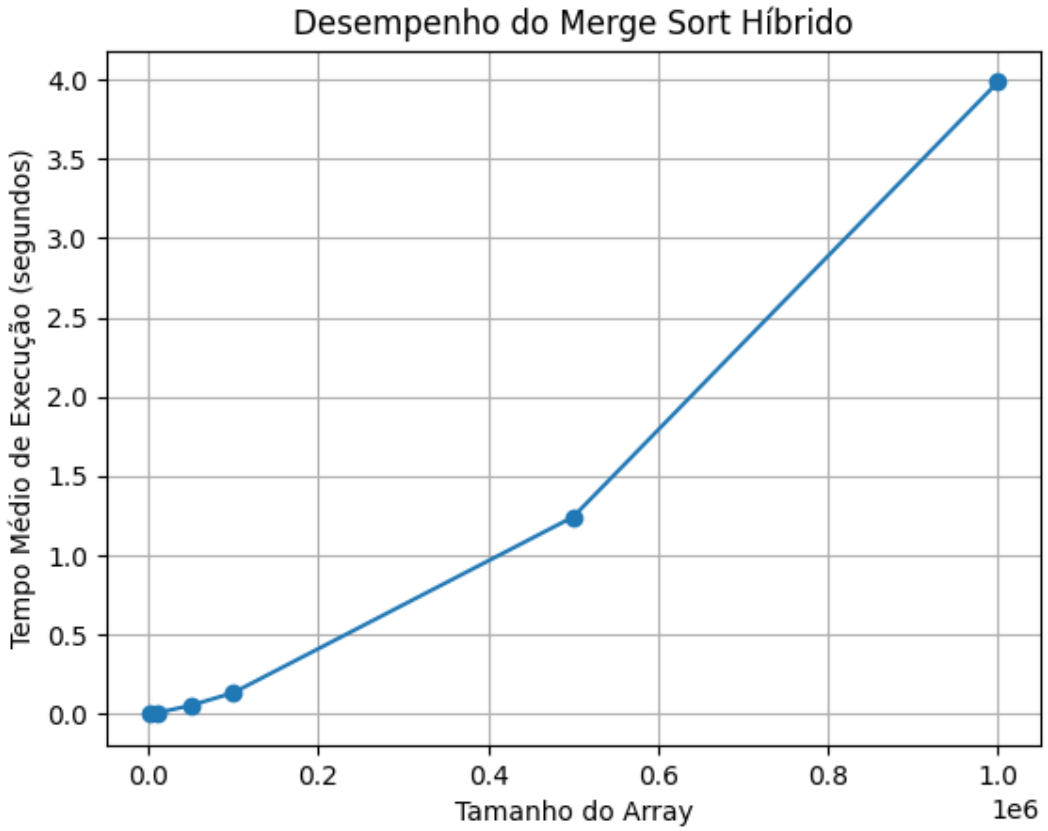
O Merge Sort Híbrido combina diferentes algoritmos de ordenação para otimizar o desempenho em diferentes situações. Ao invés de dividir o array em subarrays muito pequenos, o Merge Sort Híbrido utiliza uma estratégia que alterna para um algoritmo de ordenação mais eficiente, como o Insertion Sort, quando os subarrays atingem um tamanho específico. Essa abordagem híbrida melhora significativamente o desempenho geral do algoritmo, tornando-o mais eficiente em uma ampla variedade de tamanhos de arrays.

C1 -> Avaliação de expressão booleana (len(arr) <= 16) = CN+1
C2 -> Atribuição (meio = len(arr)) = CN
C3 -> Operação aritmética (len(arr) // 2) = CN
C4 -> Atribuição (esquerda = arr[]) = C
C5 -> Atribuição (direita = arr[]) = C
C6 -> Avaliação de expressão booleana (i < len(esquerda)) = CN
C7 -> Avaliação de expressão booleana (j < len(direita)) = CN
C8 -> Avaliação de expressão booleana(esquerda[i] < direita[j]) = CN+1
C9 -> Atribuição (arr[k] = esquerda[i]) = C
C10 -> Atribuição (i += 1) = C
C11 -> Atribuição (arr[k] = direita[j]) = C

C12 -> Atribuição (j += 1) = C
C13 -> Atribuição (k += 1) = C

C1(N+1)+C2(N)+C3(N)+C4+C5+C6(N)+C7(N)+C8(N+1)+C9+C10+C11+C12+C13

9C+6CN



Quick Sort

O Quick Sort é o algoritmo de ordenação baseado na ideia de dividir e conquistar, onde ele seleciona um elemento como pivô e divide o array em volta do pivô colocando ele na posição correta dentro do array. A função principal do Quick Sort é o particionamento que o objetivo dele é colocar os números menores para a esquerda e os números maiores na direita, ele é feito de maneira recursiva com os números de cada lado.

C1 -> Avaliação de expressão booleana (len(arr) <=1) = CN
C2 -> Retorno = CN
C3 -> Atribuição (pivo = arr[0]) = CN
C4 -> Atribuição (menores = []) = C
C5 -> Avaliação de expressão booleana (x <= pivo) = C
C6 -> Atribuição (maiores = []) = C

C7 -> Avaliação de expressão booleana (x > pivo) = C
C8 -> Retorno (return quicksort) = CN
C9 -> Operação aritmética (quicksort(menores)+[pivo]+quicksort(maiores)) = CN

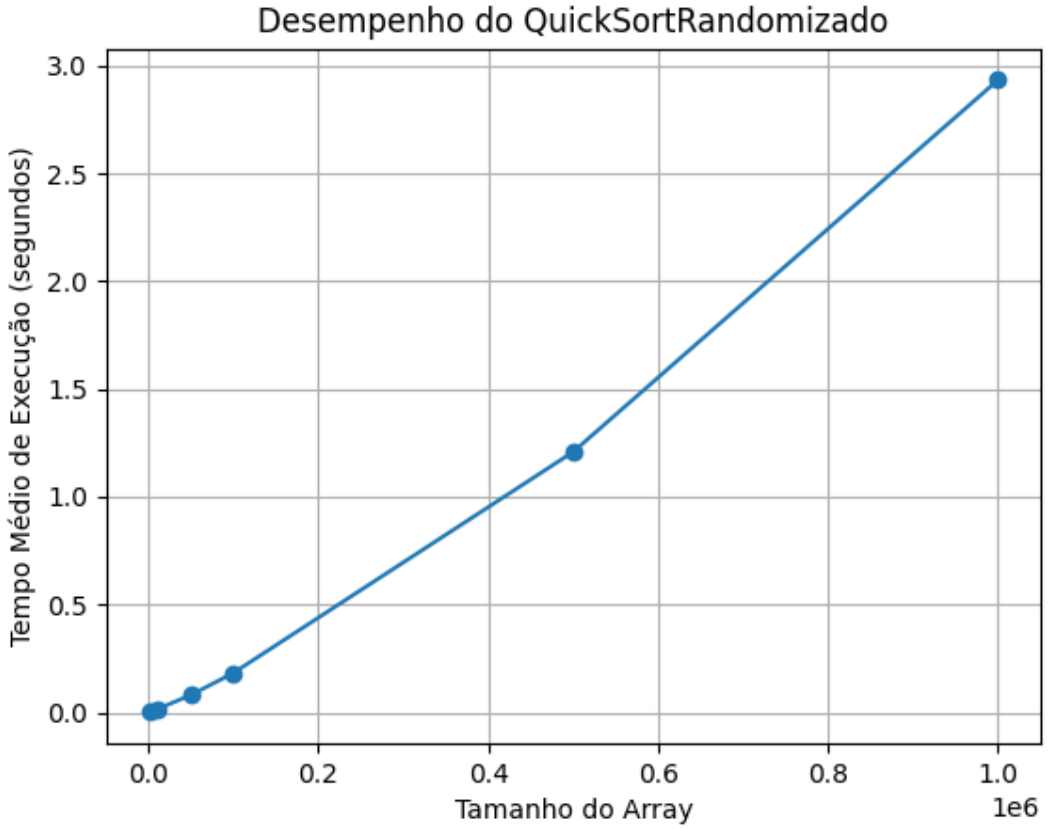
C1(N)+C2(N)+C3(N)+C4+C5+C6+C7+C8(N)+C9(N)
4C+5CN

Quick Sort Randomizado

O Quick Sort Randomizado seleciona um elemento aleatório do array como pivô, reduzindo a probabilidade de partições desbalanceadas e melhorando o desempenho médio do algoritmo. Essa escolha aleatória distribui efetivamente os elementos ao longo do processo de ordenação, evitando partições desproporcionais e proporcionando um bom desempenho em diferentes cenários de dados.

C1 -> Avaliação de expressão booleana (len(arr) <= 1) = CN+1
C2 -> Retorno (return arr) = C
C3 -> Atribuição (pivot = random.choice(arr)) = CN
C4 -> Atribuição (menores = []) = CN
C5 -> Avaliação de expressão booleana (x < pivot) = C
C6 -> Atribuição (iguais = []) = CN
C7 -> Avaliação de expressão booleana (x == pivot) = C
C8 -> Atribuição (maiores = []) = CN
C9 -> Avaliação de expressão booleana (x > pivot) = C
C10 -> Retorno (return quicksort randomizado) = C
C11 -> Operação aritmética (quicksort randomizado(menores) + iguais + quicksort randomizado(maiores)) = C

C1(N+1)+C2+C3(N)+C4(N)+C5+C6(N)+C7+C8(N)+C9+C10+C11
7C+5CN



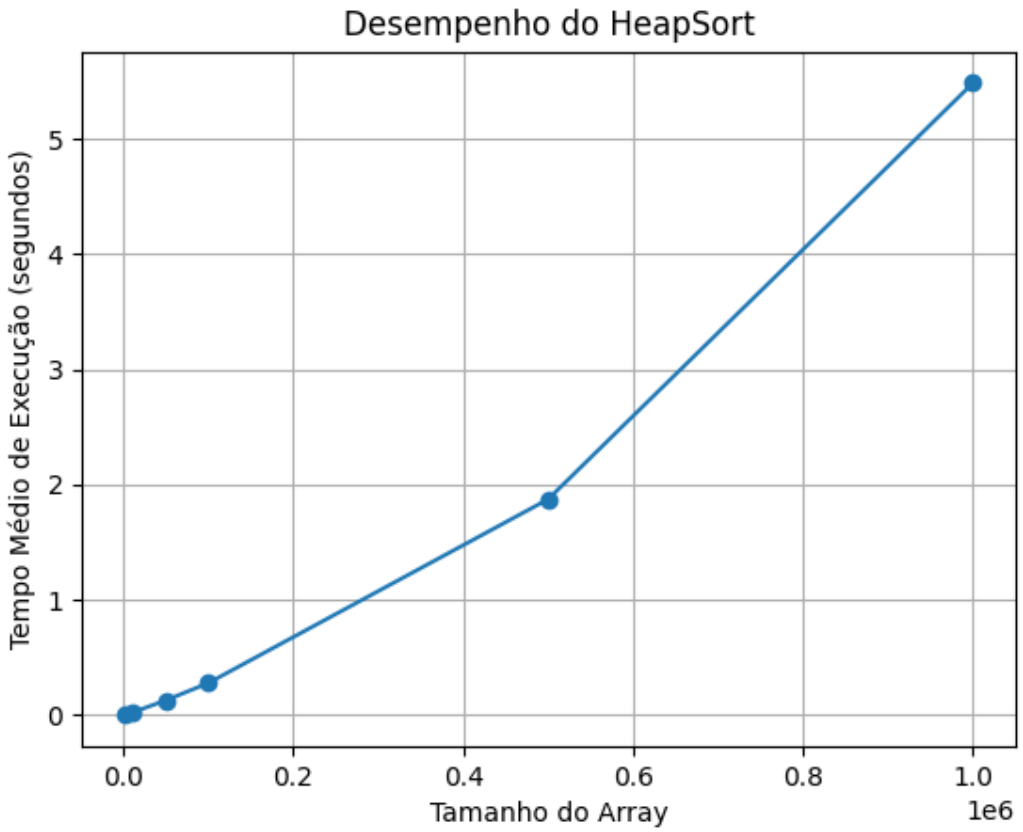
Heap Sort

O Heap Sort é o algoritmo baseado em comparações, é similar ao Selection Sort porém ele funciona transformando o array em um heap que é uma estrutura de dados como uma árvore binária completa. Sua complexidade é de O(log (n)) e apesar de não ser eficiente com dados muito complexos ele não requer muita memória e não utiliza recursão.

C1 -> Atribuição (maior = i) = C
C2 -> Atribuição (esquerda = 2 * i + 1) = C
C3 -> Operação aritmética (2 * i + 1) = C
C4 -> Atribuição (direita =2 * i + 2) = C
C5 -> Operação aritmética (2 * i + 2) = C
C6 -> Avaliação de expressão booleana (esquerda < n) = CN+1
C7 -> Avaliação de expressão booleana (arr[i] < arr[esquerda) = CN+1
C8 -> Atribuição (maior = esquerda) = C
C9 -> Avaliação de expressão booleana (direita < n) = CN+1
C10 -> Avaliação de expressão booleana (arr[maior] < arr[direita) = CN+1
C11 -> Atribuição (maior = direita) = C
C12 -> Avaliação de expressão booleana (maior != i) = CN
C13 -> Atribuição (arr[] = arr[]) = C
C14 -> Atribuição (tamanho = len(arr)) = C

C1+C2+C3+C4+C5+C6(N+1)+C7(N+1)+C8+C9(N+1)+C10(N+1)+C11+C12(N)+C13+C14

13C+5CN



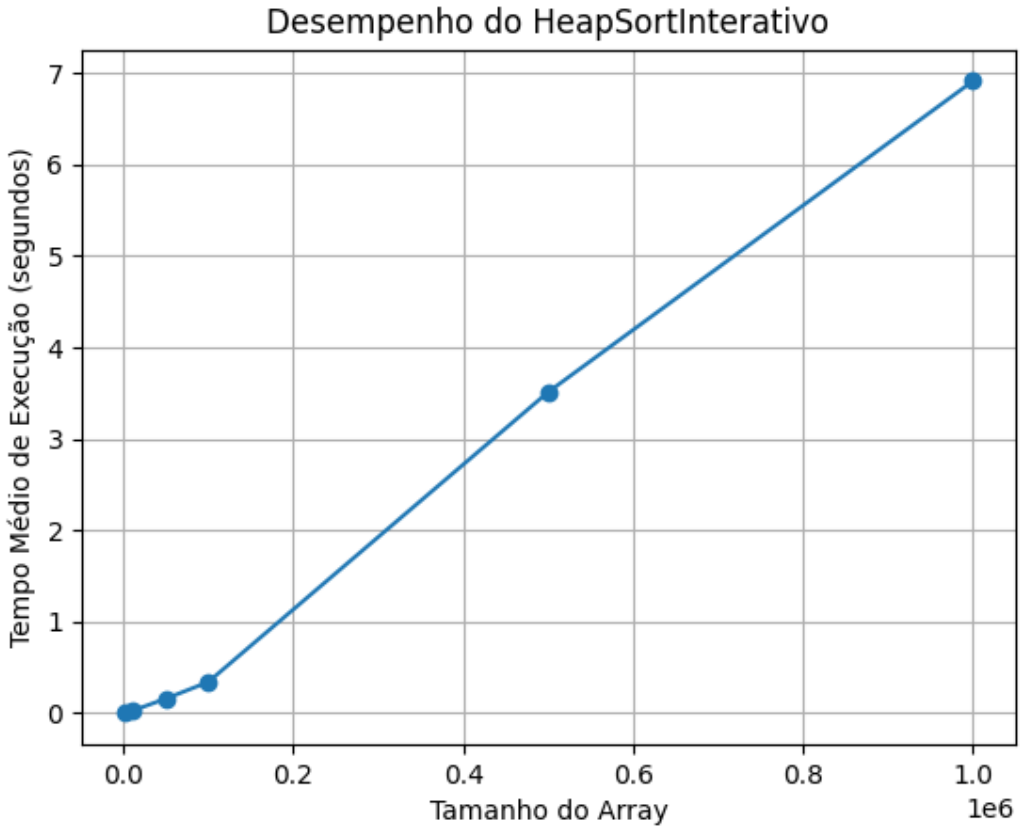
Heap Sort Iterativo

O Heap Sort Iterativo constrói o heap apenas uma vez no início do processo de ordenação, evitando a construção repetida a cada iteração da extração de um elemento. Essa melhoria reduz o tempo de execução adicional associado à construção do heap, tornando o Heap Sort mais eficiente, especialmente para arrays grandes e diversificados.

C1 -> Atribuição (n=len(arr)) = C
C2 -> Atribuição (maior = i) = C
C3 -> Atribuição (esquerda = 2 * i + 1) = C
C4 -> Operação aritmética (2 * i + 1) = C
C5 -> Atribuição (direita = 2 * i + 2) = C
C6 -> Operação aritmética (2 * i + 2) = C
C7 -> Avaliação de expressão booleana (esquerda < n) = CN
C8 -> Avaliação de expressão booleana (arr[esquerda] > arr[maior]) = CN
C9 -> Atribuição (maior = esquerda) = C
C10 -> Avaliação de expressão booleana (direita < n) = CN

C8 -> Avaliação de expressão booleana (arr[direita] > arr[maior]) = CN
C9 -> Atribuição (maior = direita) = C
C10 -> Avaliação de expressão booleana (maior != i) = CN

C1+C2+C3+C4+C5+C6+C7(N)+C8(N)+C9+C10(N)
8C+5CN



Análise

	HEAP_SORT	HEAP_ITERATIVO	MERGE_HIBRIDO	MERGE_SORT	QUICK_RANDOMIZADO
1000	0:00:00.001030	0:00:00.001545	0:00:00.000182	0:00:00.001146	0:00:00.000942
10000	0:00:00.021261	0:00:00.024401	0:00:00.007542	0:00:00.012888	0:00:00.013493
50000	0:00:00.128631	0:00:00.158325	0:00:00.054291	0:00:00.079276	0:00:00.079565
100000	0:00:00.278251	0:00:00.340884	0:00:00.134330	0:00:00.181716	0:00:00.181781
500000	0:00:01.873694	0:00:03.511785	0:00:01.242825	0:00:01.582268	0:00:01.208026
1000000	0:00:05.485806	0:00:06.914880	0:00:03.984722	0:00:04.875262	0:00:02.933863

	Bubble Sort	Bubble_Otimizado	Insertion Sort	Insertion Otimizado	Selection Sort	Selection Otimizado
1.000	00:00:00:04	00:00:00:14	00:00:00:08	00:00:00:01	00:00:00:04	00:00:00:03
5.000	00:00:10:09	00:00:00:75	00:00:00:10	00:00:00:05	00:00:00:08	00:00:00:05
10.000	00:00:43:11	00:00:02:01	00:00:00:42	00:00:00:09	00:00:00:10	00:00:00:24
50.000	00:01:50:68	00:01:04:03	00:00:10:29	00:00:01:09	00:00:08:45	00:00:06:03
100.000	00:07:13:06	00:04:09:05	00:00:41:03	00:00:04:26	00:00:35:50	00:00:24:11
500.000	07:32:51:50	03:33:14:32	00:34:24:31	00:00:03:34	00:29:07:34	00:20:01:21
1.000.000	30:17:17:96	13:49:51:01	02:19:23:23	00:00:14:26	01:54:02:03	01:23:19:12

Conclusão

Cada um dos algoritmos de ordenação melhorados apresenta vantagens específicas. O Merge Sort Híbrido é adaptável e eficiente em uma ampla gama de tamanhos de arrays, especialmente quando há uma quantidade significativa de elementos parcialmente ordenados. O Quick Sort Randomizado lida com o pior caso de desempenho do Quick Sort tradicional, tornando-o uma escolha sólida para dados desordenados ou parcialmente aleatórios. O Heap Sort Iterativo evita a construção repetida do heap, sendo eficiente para arrays grandes e diversificados. A escolha do algoritmo adequado dependerá das características dos dados e das restrições de desempenho.

Bibliografia

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

https://www.geeksforgeeks.org/quick-sort/
https://www.geeksforgeeks.org/heap-sort/
https://www.geeksforgeeks.org/merge-sort/
https://www.programiz.com/dsa/bubble-sort
https://www.programiz.com/dsa/selection-sort
https://www.programiz.com/dsa/insertion-sort
https://www.programiz.com/dsa/merge-sort
https://www.programiz.com/dsa/quick-sort
https://www.programiz.com/dsa/heap-sort
https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/OBubbleSort.html
https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/AOrdenacaoPorSelecao.html
https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/AOrdenacaoPorInsercao.html
https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/OMergeSort.html
https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/OQuickSort.html
https://www.javatpoint.com/bubble-sort-in-python
https://www.javatpoint.com/selection-sort-in-python
https://www.javatpoint.com/insertion-sort-in-python
https://www.javatpoint.com/merge-sort-in-python
https://www.javatpoint.com/heap-sort-in-python
https://www.javatpoint.com/quicksort-in-python