Complexidade de algoritmos de ordenação

Iori Fernando Carvalho Rodrigues[1]

[1]Departamento de engenharia de computação – Universidade Estadual do Maranhão (UEMA)
Cidade Universitária Paulo VI 1, São Luís, MA, 65055-310

iorirodrigues@aluno.uema.br

Abstract. *This paper presents a detailed computational complexity analysis of three sorting algorithms: Cocktail Shaker Sort, Comb Sort, and Cycle Sort. The aim is to understand the behavior of these algorithms through a theoretical approach involving line-by-line decomposition of their structures, construction of corresponding algebraic expressions, and deduction of their asymptotic notation (Big-O). The adopted methodology enables the identification of the computational cost of each operation performed, as well as the complexity in best, worst, and average cases. The algorithms were compared in terms of time efficiency and memory usage. Theoretical results indicate that although all of them have quadratic complexity in the worst case, Comb Sort stands out for its better average-case performance. It is concluded that a formal complexity analysis is essential for the appropriate selection of algorithms in various contexts, especially when scalability is a concern.*

Resumo. *Este trabalho apresenta uma análise detalhada da complexidade computacional de três algoritmos de ordenação: Cocktail Shaker Sort, Comb Sort e Cycle Sort. O objetivo é compreender o comportamento desses algoritmos por meio de uma abordagem teórica que envolve a decomposição de suas estruturas linha a linha, a montagem das expressões algébricas correspondentes e a dedução da notação assintótica (Big-O). A metodologia adotada permite identificar o custo computacional de cada operação realizada, além de calcular a complexidade nos melhores, piores e casos médios. Os algoritmos foram comparados quanto à sua eficiência temporal e ao uso de memória. Os resultados teóricos indicam que, embora todos apresentem complexidade quadrática no pior caso, o Comb Sort se destaca por apresentar melhor desempenho médio. Conclui-se que a análise formal da complexidade é essencial para a escolha apropriada de algoritmos em diferentes contextos, especialmente quando se considera a escalabilidade das soluções.*

1. Introdução

A análise de complexidade de algoritmos é um dos pilares da ciência da computação teórica, pois permite estimar o desempenho de programas e estruturas de dados sem a necessidade de execução prática. Em especial, algoritmos de ordenação são fundamentais em diversas aplicações, sendo recorrente a necessidade de avaliá-los quanto à eficiência temporal e espacial.

O objetivo deste trabalho é analisar detalhadamente a complexidade dos algoritmos de ordenação *Cocktail Shaker Sort*, *Comb Sort* e *Cycle Sort*, destacando suas estruturas e deduzindo matematicamente sua complexidade computacional.

A metodologia empregada consiste na análise linha a linha de cada algoritmo, com a dedução algébrica da função de complexidade, seguida de sua resolução e simplificação até a notação assintótica (*Big-O*).

2. Fundamentação Teórica

2.1. Algoritmos de Ordenação

Os algoritmos de ordenação são procedimentos fundamentais na ciência da computação, aplicados em inúmeras tarefas como busca eficiente, visualização de dados e otimização de sistemas. Seu objetivo é reorganizar os elementos de uma estrutura de dados — comumente vetores ou listas — em uma determinada ordem, geralmente crescente ou decrescente.

Existem algoritmos elementares, como o *Bubble Sort*, que são didaticamente úteis, mas apresentam desempenho ineficiente em grandes volumes de dados. Por outro lado, algoritmos como *Merge Sort*, *Quick Sort* e os abordados neste trabalho (*Cocktail Shaker Sort*, *Comb Sort*, *Cycle Sort*) apresentam estratégias mais sofisticadas de comparação, particionamento e movimentação de elementos, com impactos diretos na eficiência computacional.

2.2. Complexidade de Algoritmos

A complexidade de um algoritmo refere-se à quantidade de recursos computacionais que ele demanda em função do tamanho da entrada, denotada por n. Essa análise pode ser dividida em duas categorias principais:

- **Complexidade Temporal**: mede o número de operações (ou passos) que o algoritmo executa até sua conclusão.
- **Complexidade Espacial**: avalia a quantidade de memória adicional necessária durante a execução.

A notação assintótica, como a notação Big-O ($\mathcal{O}$), é utilizada para expressar o comportamento da complexidade quando n tende ao infinito. Ela permite classificar algoritmos independentemente da arquitetura de hardware ou linguagem de programação utilizada.

A análise pode considerar três cenários distintos:

- **Melhor caso**: desempenho sob as condições mais favoráveis.
- **Pior caso:** desempenho sob as condições mais desfavoráveis.
- **Caso médio**: desempenho esperado com entradas aleatórias.

A dedução da complexidade é feita a partir da contagem das operações dominantes do algoritmo, buscando sua função de crescimento mais representativa. Essa análise é essencial para selecionar o algoritmo mais adequado a uma aplicação prática, especialmente em sistemas com restrições de tempo ou memória.

Para calcular a notação Big-O, identificamos quantas vezes as partes mais importantes do algoritmo (geralmente os laços ou recursões) são executadas. Por exemplo, em um algoritmo que tem um laço duplo como:

```
for i in range(n):
    for j in range(n):
        fazer_algo()
```

A função `fazer_algo()` é executada $n \times n = n^2$ vezes. Por isso, dizemos que esse algoritmo tem complexidade $\mathcal{O}(n^2)$. Mesmo que o número exato de operações fosse $3n^2 + 2n + 5$, usamos apenas o termo de maior crescimento (n^2) na notação Big-O.

3. Análise Detalhada dos Algoritmos

3.1. Cocktail Shaker Sort

O *Cocktail Shaker Sort*, também conhecido como *Bidirectional Bubble Sort*, é uma variação aprimorada do algoritmo *Bubble Sort* tradicional. Sua principal característica é a realização de varreduras em ambas as direções da lista: da esquerda para a direita e, em seguida, da direita para a esquerda, dentro de um mesmo ciclo de iteração.

No *Bubble Sort*, os maiores elementos "borbulham" até o final da lista por meio de trocas sucessivas. No entanto, esse processo pode ser ineficiente para elementos pequenos posicionados no final da lista. O *Cocktail Shaker Sort* resolve esse problema realizando uma varredura reversa, permitindo que elementos menores retornem mais rapidamente para o início da sequência. Isso reduz o número de ciclos necessários em listas parcialmente ordenadas.

O algoritmo utiliza um indicador lógico para controlar se houve alguma troca durante uma varredura. Se nenhuma troca ocorrer em um ciclo completo (ida e volta), considera-se que a lista está ordenada e o processo é encerrado. Além disso, os limites inicial e final da varredura são ajustados a cada iteração, restringindo as comparações apenas à região onde ainda há elementos possivelmente desordenados.

Essa abordagem torna o algoritmo mais eficiente que o *Bubble Sort* em alguns casos práticos, especialmente quando a desordem está concentrada nas extremidades da lista. Entretanto, sua complexidade assintótica no pior caso permanece quadrática, ou seja, $O(n^2)$, semelhante à de algoritmos de ordenação simples baseados em comparação.

3.1.1. Código

```
def ordenacao_cycle_sort(lista):
    n = len(lista)
    for inicio_ciclo in range(0, n - 1):
        item = lista[inicio_ciclo]
        pos = inicio_ciclo
        for i in range(inicio_ciclo + 1, n):
            if lista[i] < item:
                pos += 1
        if pos == inicio_ciclo:
            continue
        while item == lista[pos]:
            pos += 1
```

```
    lista[pos], item = item, lista[pos]
    while pos != inicio_ciclo:
        pos = inicio_ciclo
        for i in range(inicio_ciclo + 1, n):
            if lista[i] < item:
                pos += 1
        while item == lista[pos]:
            pos += 1
        lista[pos], item = item, lista[pos]
```

3.1.2. Análise de Complexidade

Linha	Código	Descrição	Custo
1	`while trocou:`	Laço principal até ordenação	$\leq O(n)$
2	`for i in range(...)`	Laços de ida e volta	$O(n)$ cada
3	`if lista[i] > lista[i+1]`	Comparação entre pares	$O(1)$
4	`Troca condicional`	Troca de elementos adjacentes	$O(1)$

Table 1. Análise simplificada de complexidade do Cocktail Shaker Sort

Cada passagem (ida e volta) realiza até $2(n-1)$ comparações. No pior caso, podem ser necessárias até $n/2$ passagens. Assim:

$$T(n) = \frac{n}{2} \cdot 2(n-1) = n(n-1) \Rightarrow O(n^2)$$

3.1.3. Resumo da Complexidade:

- Melhor caso: $O(n)$ (lista já ordenada).
- Pior caso: $O(n^2)$.
- Caso médio: $O(n^2)$.

3.2. Comb Sort

O *Comb Sort* é um algoritmo de ordenação baseado no *Bubble Sort*, mas com uma estratégia otimizada para eliminar elementos pequenos que ficam ”presas” no final da lista — um problema típico do algoritmo original. Para isso, o *Comb Sort* introduz o conceito de ”intervalo” (*gap*) entre os elementos comparados, permitindo trocas entre elementos distantes.

Inicialmente, o intervalo é igual ao tamanho da lista, e a cada iteração ele é reduzido por um fator fixo (comumente 1,3). A redução continua até que o intervalo se torne igual a 1, momento em que o algoritmo se comporta como um *Bubble Sort* tradicional. Essa estratégia permite uma reorganização mais eficiente dos elementos no início do processo, tornando o algoritmo mais rápido que o *Bubble Sort*, principalmente em listas maiores.

O algoritmo utiliza uma variável booleana para verificar se a lista já está ordenada. Se, após uma passagem com intervalo 1, nenhuma troca for realizada, considera-se que a lista está ordenada, e o processo é encerrado. Essa abordagem permite uma melhora significativa no desempenho médio do algoritmo, embora, no pior caso, a complexidade continue sendo quadrática.

Por não utilizar recursão nem estruturas auxiliares, o *Comb Sort* é uma boa escolha em sistemas com recursos limitados, sendo especialmente útil quando é importante reduzir o número de comparações nas primeiras fases da ordenação.

3.2.1. Código

```
def ordenacao_comb_sort(lista):
    n = len(lista)
    intervalo = n
    reducao = 1.3
    ordenado = False
    while not ordenado:
        intervalo = int(intervalo / reducao)
        if intervalo <= 1:
            intervalo = 1
            ordenado = True
        i = 0
        while i + intervalo < n:
            if lista[i] > lista[i + intervalo]:
                lista[i], lista[i + intervalo] = lista[i + intervalo],
                ordenado = False
            i += 1
```

3.2.2. Análise de Complexidade

Linha	Código	Descrição	Custo
1	`while not ordenado:`	Laço principal até ordenação	$\leq O(\log n)$
2	`intervalo = intervalo / 1.3`	Redução do intervalo	$O(1)$
3	`while i + intervalo < n:`	Percorre lista com intervalo	$O(n)$
4	`if lista[i] > lista[i + intervalo]`	Comparação entre pares	$O(1)$
5	`Troca condicional`	Troca de elementos distantes	$O(1)$

Table 2. Análise simplificada de complexidade do Comb Sort

Como o intervalo é reduzido de forma exponencial, o número de ciclos principais é proporcional a $\log_{1.3} n$, e cada ciclo pode realizar até n comparações. Assim:

$$T(n) = O(n \log n)$$

No entanto, se a lista exigir muitas trocas mesmo com intervalos pequenos, o número total de operações pode se aproximar de $O(n^2)$ no pior caso.

3.2.3. Resumo da Complexidade:

- Melhor caso: $O(n \log n)$.
- Pior caso: $O(n^2)$.
- Caso médio: $O(n \log n)$.

3.3. Cycle Sort

O *Cycle Sort* é um algoritmo de ordenação in-place, que se destaca por minimizar o número de movimentações de elementos. Sua estratégia consiste em identificar ciclos de permutação na lista e reposicionar cada elemento diretamente em sua posição final, sempre que possível. Esse comportamento é útil, por exemplo, quando o custo de escrita em memória é elevado (como em EEPROMs ou flash).

O algoritmo percorre a lista uma vez, iniciando um novo ciclo de reposicionamento sempre que encontra um elemento fora de lugar. Ele calcula a posição correta do elemento atual contando quantos elementos menores que ele existem à frente. Se o valor estiver fora do lugar, ele é trocado diretamente com o elemento que está em sua posição correta. Esse processo se repete até que o ciclo seja fechado.

Apesar do número de trocas ser mínimo ($O(n)$ no pior caso), o número de comparações pode ser alto, pois o algoritmo precisa encontrar a posição correta de cada elemento e lidar com duplicatas. Assim, a complexidade total em tempo continua sendo quadrática, embora o algoritmo seja eficiente em termos de escrita.

3.3.1. Código

```
def ordenacao_cycle_sort(lista):
    n = len(lista)
    for inicio_ciclo in range(0, n - 1):
        item = lista[inicio_ciclo]
        pos = inicio_ciclo
        for i in range(inicio_ciclo + 1, n):
            if lista[i] < item:
                pos += 1
        if pos == inicio_ciclo:
            continue
        while item == lista[pos]:
            pos += 1
        lista[pos], item = item, lista[pos]
        while pos != inicio_ciclo:
            pos = inicio_ciclo
            for i in range(inicio_ciclo + 1, n):
                if lista[i] < item:
```

```
            pos += 1
    while item == lista[pos]:
        pos += 1
    lista[pos], item = item, lista[pos]
```

3.3.2. Análise de Complexidade

Linha	Código	Descrição	Custo
1	`for inicio in range(n - 1):`	Laço principal por posição inicial	$O(n)$
2	`for i in range(...):`	Conta quantos elementos são menores	$O(n)$
3	`while item == lista[pos]:`	Lida com elementos duplicados	$O(n)$
4	`while pos != inicio:`	Recomeça o ciclo se ainda não fechado	$O(n)$
5	`Troca condicional`	Coloca item em sua posição correta	$O(1)$

Table 3. Análise simplificada de complexidade do Cycle Sort

Cada ciclo pode exigir até n comparações para determinar a posição correta, e o número de ciclos também pode chegar a n. Assim:

$$T(n) = O(n^2)$$

Em contrapartida, o número de trocas realizadas é mínimo e limitado por $O(n)$, pois cada elemento é movido no máximo uma vez para sua posição final. Isso torna o *Cycle Sort* especialmente eficiente em contextos nos quais a movimentação física de dados tem custo elevado. Por essa razão, embora o algoritmo não seja o mais rápido em termos de tempo de execução, ele se destaca por sua eficiência em termos de número de escritas — o menor possível entre todos os algoritmos de ordenação baseados em comparação.

3.3.3. Resumo da Complexidade:

- Melhor caso: $O(n^2)$.
- Pior caso: $O(n^2)$.
- Caso médio: $O(n^2)$.

4. Comparação Entre os Algoritmos

Algoritmo	Melhor Caso	Pior Caso	Caso Médio	Complexidade Espacial
Cocktail Shaker Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Comb Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(1)$
Cycle Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

Table 4. Comparação teórica entre os algoritmos analisados.

Cada algoritmo apresenta características que o tornam mais adequado dependendo do contexto de aplicação:

- O *Cocktail Shaker Sort* é eficiente para listas parcialmente ordenadas, aproveitando a quase ordenação inicial.
- O *Comb Sort* apresenta bom desempenho no caso médio, sendo geralmente mais rápido que o *Bubble Sort*.
- O *Cycle Sort*, apesar de sua complexidade temporal elevada, é especialmente útil quando é necessário minimizar o número de movimentações de elementos.

4.1. Ensaios Experimentais

Para validar as análises teóricas, realizamos ensaios experimentais medindo os seguintes parâmetros:

- **Tempo de execução**: duração da ordenação, em segundos, excluindo inicializações e operações de entrada e saída. As entradas testadas foram $n = \{100, 1\,000, 10\,000, 100\,000, 1\,000\,000, 10\,000\,000\}$.
- **Consumo de memória**: uso de memória durante a execução do algoritmo, em megabytes (MB).

  Os resultados foram armazenados em um arquivo CSV com a seguinte estrutura:

```
Data:                                   AAAA-MM-DD
Hora:                                   HH-MM-SS
Algoritmo, N, Caso, Tempo, Memória
```

Exemplo de linha do arquivo CSV:

2025-06-09, 17:50:13, Comb Sort, 10000, Caso médio, 112.134, 1024

Os dados completos dos ensaios estão disponíveis em anexo e foram utilizados para a análise a seguir.

4.2. Análise Experimental dos Algoritmos

Agora seram apresentados os resultados dos ensaios experimentais realizados para avaliar o desempenho dos algoritmos de ordenação estudados. Foram gerados gráficos comparativos que ilustram o tempo de execução dos algoritmos nos três principais cenários: melhor caso, pior caso e caso médio. Esses gráficos possibilitam uma visualização clara e direta das diferenças de eficiência entre o *Cocktail Shaker Sort*, *Comb Sort* e *Cycle Sort* à medida que o tamanho da entrada aumenta. Além disso, a análise desses dados experimentais complementa a análise teórica previamente discutida, permitindo uma compreensão mais aprofundada do comportamento prático de cada algoritmo sob condições variadas. Esta abordagem integrada entre teoria e prática é fundamental para a escolha consciente do algoritmo mais adequado para diferentes situações.

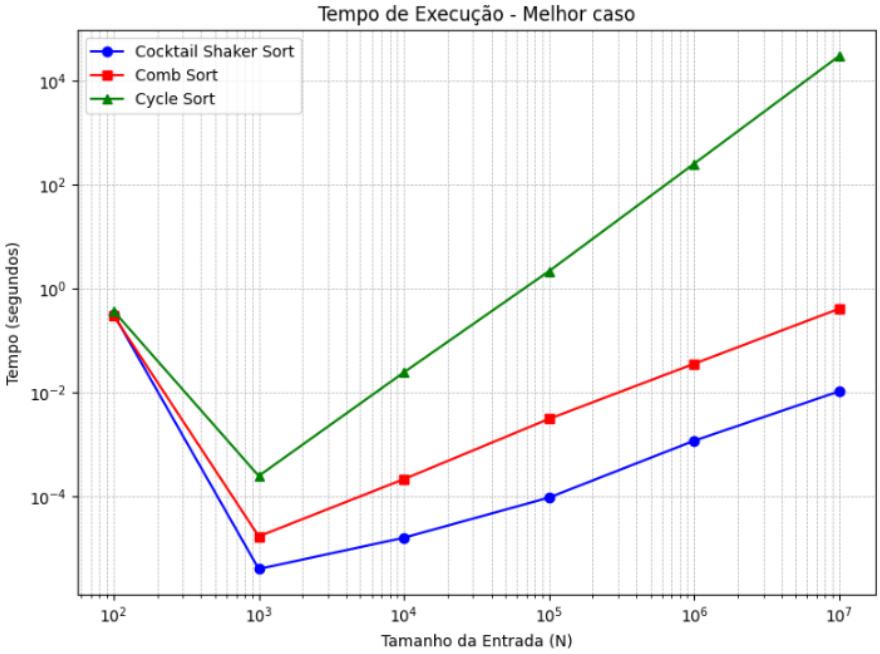

Figure 1. Tempo de execução no melhor caso para os algoritmos Cocktail Shaker Sort, Comb Sort e Cycle Sort.

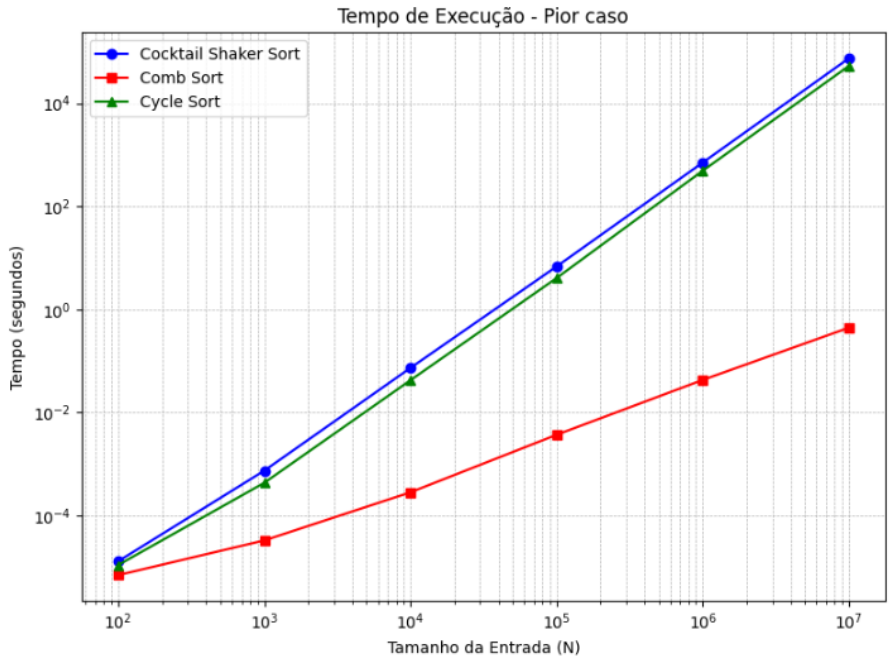

Figure 2. Tempo de execução no pior caso para os algoritmos Cocktail Shaker Sort, Comb Sort e Cycle Sort.

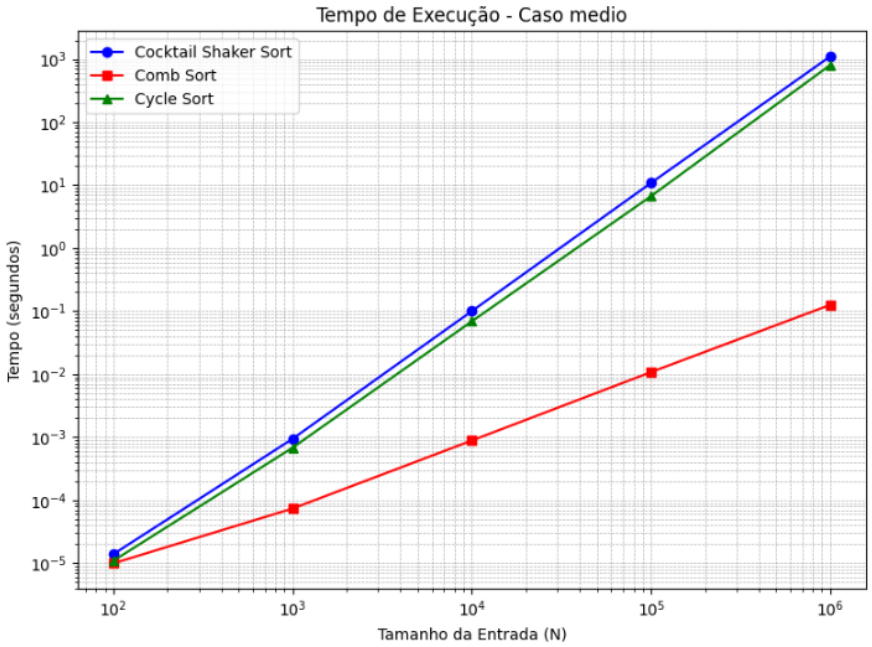

Figure 3. Tempo de execução no caso médio para os algoritmos Cocktail Shaker Sort, Comb Sort e Cycle Sort.

5. Conclusão

Este trabalho apresentou uma análise detalhada da complexidade computacional de três algoritmos de ordenação, por meio de decomposição linha a linha e dedução matemática. Observou-se que, apesar de apresentarem complexidade quadrática no pior caso, suas eficiências variam significativamente conforme o cenário de entrada.

A compreensão dessas características permite a escolha mais adequada de algoritmos para diferentes problemas, considerando não apenas o tempo de execução, mas também o consumo de memória e a natureza da entrada.

Como continuidade, recomenda-se a realização de análises empíricas adicionais utilizando diferentes tipos e tamanhos de dados para aprofundar a avaliação prática dos algoritmos.

6. Referências Bibliográficas

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

SEDGEWICK, Robert. *Algorithms in C*. Addison Wesley, 1998.

GITHUB. TheAlgorithms / Python. Disponível em: `https://github.com/TheAlgorithms/Python/blob/master/sorts/cocktail_shaker_sort.py`. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Algoritmo*. Disponível em: `https://pt.wikipedia.org/wiki/Algoritmo`. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Análise de algoritmos*. Disponível em: `https://pt.wikipedia.org/wiki/Anlise_de_algoritmos`. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Algoritmo de ordenação*. Disponível em: https://pt.wikipedia.org/wiki/Algoritmo_de_ordenacao. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Cocktail shaker sort*. Disponível em: https://en.wikipedia.org/wiki/Cocktail_shaker_sort. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Comb sort*. Disponível em: https://pt.wikipedia.org/wiki/Comb_sort. Acesso em: 12 jun. 2025.

WIKIPÉDIA. *Cycle sort*. Disponível em: https://en.wikipedia.org/wiki/Cycle_sort. Acesso em: 12 jun. 2025.