# Avaliação Teórica e Experimental dos Algoritmos de Ordenação Shell Sort e Cycle Sort

**Luciano Sousa Barbosa**[1], **Pedro Henrique Silva Rodrigues**[1], **Tiago Lima de Moura**[1]

[1]Departamento de Sistemas de Informação – Universidade Federal do Piauí (UFPI)
64.600-000 – Picos – PI – Brasil

{luciano.barbosa,pedro.rodrigues.pr,tiago.lima}@ufpi.edu.br

***Resumo.*** *Este trabalho apresenta uma análise comparativa dos algoritmos de ordenação Shell Sort e Cycle Sort, considerando aspectos teóricos e experimentais. Os algoritmos foram implementados na linguagem C e avaliados sob diferentes configurações de entrada, variando o tamanho dos vetores e o grau de ordenação inicial dos dados. A análise experimental considerou múltiplas métricas e medidas estatísticas. Os resultados obtidos possibilitam avaliar o comportamento prático dos algoritmos e relacioná-los com a complexidade assintótica. Por fim, são discutidas as diferenças de desempenho observadas e a sensibilidade de cada algoritmo à disposição inicial dos elementos.*

## 1. Introdução

A ordenação de dados é uma operação fundamental em Ciência da Computação, estando presente em aplicações como processamento de informações, bancos de dados e sistemas de recuperação de dados [Thomas H et al. 2009]. A eficiência dos algoritmos de ordenação influencia diretamente o desempenho global desses sistemas, especialmente quando o volume de dados cresce significativamente [Sedgewick and Wayne 2011]. Dessa forma, o estudo sistemático desses algoritmos torna-se essencial para compreender os limites e aplicações práticas em diferentes contextos computacionais [Donald et al. 1997].

Diferentes algoritmos de ordenação apresentam comportamentos distintos conforme a organização inicial dos dados de entrada e o conjunto de operações realizadas internamente [Bentley and McIlroy 1993]. Alguns algoritmos buscam minimizar o número de comparações, enquanto outros priorizam a redução no número de trocas, o que resulta em desempenhos variados dependendo do cenário analisado [Sundaramoorthy and Karunanidhi 2025]. Nesse sentido, a análise exclusivamente teórica nem sempre é suficiente para caracterizar o desempenho em ambientes reais de execução [Hopcroft et al. 1983].

Entre os algoritmos baseados em comparação, o *Shell Sort* e o *Cycle Sort* apresentam características estruturais distintas que motivam sua avaliação conjunta [Sedgewick and Wayne 2011]. O *Shell Sort* estende o *Insertion Sort* por meio da utilização de incrementos, buscando reduzir o custo das movimentações de elementos ao longo do processo de ordenação [Donald et al. 1997]. Por sua vez, o *Cycle Sort* foi desenvolvido com o objetivo de minimizar o número de trocas realizadas, característica relevante em cenários onde a escrita em memória é custosa [Thomas H et al. 2009].

A análise experimental desses algoritmos permite observar como tais diferenças teóricas se manifestam na prática, especialmente sob diferentes padrões de ordenação

inicial dos dados [Ayazuddin 2025]. Métricas como tempo de execução, número de comparações, quantidade de trocas e medidas estatísticas associadas fornecem uma visão mais abrangente do comportamento dos algoritmos [Jain 1990]. Dessa forma, torna-se possível identificar discrepâncias entre o desempenho teórico e o desempenho prático observado [Muhammad et al. 2025].

Assim, o objetivo geral deste trabalho é realizar uma avaliação comparativa dos algoritmos de ordenação *Shell Sort* e *Cycle Sort*, considerando desempenho, complexidade assintótica e sensibilidade à ordenação inicial dos dados. A partir de experimentos controlados, busca-se relacionar os resultados empíricos com as previsões teóricas, contribuindo para uma compreensão mais sólida sobre a aplicação desses algoritmos em diferentes cenários computacionais.

## 2. Referencial Teórico

Nesta seção, são apresentados os fundamentos teóricos relacionados aos algoritmos de ordenação estudados neste trabalho, abordando seus princípios de funcionamento, características estruturais e particularidades operacionais. São discutidos os aspectos conceituais que orientam o comportamento dos algoritmos *Shell Sort* e *Cycle Sort*, bem como suas estratégias internas de ordenação e impactos sobre o desempenho. Além disso, são analisadas as complexidades assintóticas associadas a cada algoritmo, considerando diferentes configurações de entrada. Conceitos práticos serão introduzidos para conectar a teoria dos algoritmos e os resultados experimentais apresentados nas seções subsequentes.

### 2.1. Shell Sort

O algoritmo *Shell Sort* foi proposto por [Shell 1959] como uma generalização do *Insertion Sort*, com o objetivo de reduzir o número de deslocamentos necessários durante o processo de ordenação. Ao introduzir comparações entre elementos distantes, o algoritmo busca minimizar a influência de entradas desfavoráveis, conectando-se diretamente à limitação do *Insertion Sort* em vetores grandes e desordenados [Lipton and Naughton ].

A ideia central do *Shell Sort* consiste em ordenar o vetor por meio de subsequências geradas a partir de um intervalo (*gap*), que é progressivamente reduzido até atingir o valor unitário, momento em que o algoritmo se comporta como um *Insertion Sort* tradicional [Mahmoud 2011]. Essa estratégia permite que elementos distantes de suas posições finais sejam movidos mais rapidamente, estabelecendo uma transição eficiente entre ordenações grosseiras e refinadas [Smythe and Wellner 2001].

Um dos aspectos mais relevantes do *Shell Sort* está na escolha da sequência de incrementos (*gap sequence*), fator que influencia diretamente seu desempenho prático e teórico [Janson and Knuth 1997]. Diversas sequências foram propostas ao longo do tempo, como as de [Shell 1959, Hibbard 1962, Lipton and Naughton , Pratt 1972], cada uma apresentando impactos distintos na complexidade e no comportamento empírico do algoritmo.

Do ponto de vista da complexidade assintótica, o *Shell Sort* não possui uma análise fechada e única, uma vez que sua complexidade depende fortemente da sequência de incrementos adotada [Souza et al. 2018]. Em cenários práticos, determinadas sequências permitem desempenhos próximos de $O(n^{3/2})$ ou até melhores, enquanto no pior caso

teórico algumas configurações podem se aproximar de $O(n^2)$ [Frank and Lazarus 1960, Sedgewick 1986]. Observe a tabela 1.

**Tabela 1. Complexidade Analítica Para Algumas Sequências de Passos.**

| **Autor Ano** | **Passos** $\{p_1, \ldots, p_k\}$ | **Pior caso** |
|---|---|---|
| [Shell 1959] | $\{n/2^i : 0 < i \leq \lfloor \log_2 n \rfloor\}$ | $\Theta(n^2)$ |
| [Frank 1960] | $\{2 \cdot \lfloor n/2^i \rfloor + 1 : 0 < i \leq \lfloor \log_2 n \rfloor\}$ | $\Theta(n^{3/2})$ |
| [Hibbard 1962] | $\{2^i - 1 : 0 < i \leq \lfloor \log_2 n \rfloor\}$ | $\Theta(n^{3/2})$ |
| [Papernov 1965] | $\{1\} \cup \{2^{i-1} + 1 : 1 < i \leq \lfloor \log_2 n \rfloor\}$ | $\Theta(n^{3/2})$ |
| [Pratt 1972] | $\{q = 2^r 3^s : r, s \in \mathbb{N} \mid q < n\}$ | $\Theta(n \log^2 n)$ |
| [Knuth 1973] | $\{q = (3^i - 1)/2 : i \in \mathbb{N} \mid q \leq \lceil n/3 \rceil\}$ | $\Theta(n^{3/2})$ |
| [Sedgewick 1986] | $\{q = 4^i + 3 \cdot 2^{i-1} + 1 : 1 < i \in \mathbb{N} \mid q < n\}$ | $\Theta(n^{4/3})$ |
| [Gonnet 1991] | $\left\{(0.45454)^i \cdot n : 0 < i \leq \lfloor \log_{2/2} n \rfloor\right\}$ | em aberto |
| [Tokuda 1992] | $\left\{q = \left\lfloor \frac{9^i - 4^i}{5 \cdot 4^{i-1}} \right\rfloor : 1 < i \in \mathbb{N} \mid q < n\right\}$ | em aberto |

Em termos de características operacionais, o *Shell Sort* é um algoritmo *in-place*, não estável e de fácil implementação, o que contribui para sua ampla utilização em contextos educacionais e aplicações com restrições de memória [Smythe and Wellner 2002]. Essas propriedades o tornam especialmente interessante para estudos comparativos, pois combinam simplicidade estrutural com desempenho competitivo em entradas parcialmente ordenadas [Gonnet and Baeza-Yates 1991].

Por fim, o funcionamento interno do *Shell Sort* evidencia uma relação direta entre teoria e prática, uma vez que pequenas variações na configuração dos *gaps* podem resultar em diferenças significativas no número de comparações e trocas realizadas. Essa sensibilidade justifica a necessidade de análises experimentais complementares, conectando os fundamentos teóricos às observações empíricas apresentadas neste trabalho. A figura 1 abaixo apresenta uma implementação em pseudocódigo do algoritmo.

**Listing 1. Implementação em Pseudocódigo do Shell Sort.**

```
PROCEDIMENTO shellSort(v[], n)
    // Calcula o primeiro passo usando a sequencia de Knuth
    k = floor(log3(n + 1) + 0.5)
    h = (3^k - 1) / 2

    // Loop principal: executa para cada valor de h
    ENQUANTO h >= 1 FACA
        // Insertion sort com passo h
        PARA i = h ATE n-1 FACA
            temp = v[i] // Elemento a ser posicionado
            j = i // Indice para comparacoes

            // Move elementos maiores que temp
            ENQUANTO j >= h FACA
                // Encontra posicao correta para temp
                SE v[j-h] <= temp ENTAO PARAR
                v[j] = v[j-h] // Desloca elemento
                j = j - h // Atualiza indice
            FIM ENQUANTO

            // Insere temp na posicao correta
            v[j] = temp
```

```
        FIM PARA

        // Proximo passo da sequencia de Knuth
        h = (h - 1) / 3
    FIM ENQUANTO
FIM PROCEDIMENTO
```

## 2.2. Cycle Sort

O *Cycle Sort* é um algoritmo de ordenação baseado no conceito de ciclos de permutação, tendo sido formalmente descrito no contexto de técnicas para minimização de escritas em memória, característica que o diferencia de algoritmos clássicos de comparação [Haddon 1990]. Essa motivação histórica conecta-se diretamente a cenários onde o custo de escrita é elevado, como em memórias *EEPROM* e sistemas embarcados.

O princípio fundamental do *Cycle Sort* consiste em posicionar cada elemento diretamente em sua posição final correta, formando ciclos de movimentação que evitam trocas desnecessárias [Sedgewick and Wayne 2011]. Ao completar um ciclo, o algoritmo garante que todos os elementos envolvidos estejam corretamente posicionados, conectando a noção matemática de permutação à ordenação prática de dados.

Do ponto de vista operacional, o algoritmo percorre o vetor identificando quantos elementos são menores que o item corrente, determinando assim sua posição correta no vetor ordenado [Jain 1990]. Caso o elemento já esteja na posição adequada, o algoritmo avança, mantendo uma transição eficiente entre ciclos completos e iterações subsequentes.

Em relação à complexidade assintótica, o *Cycle Sort* apresenta complexidade de tempo $O(n^2)$ em todos os casos, uma vez que depende de comparações repetidas para determinar a posição correta de cada elemento [Turzo et al. 2020]. Entretanto, sua complexidade de escrita é ótima, realizando no máximo $n$ - 1 troca, o que o torna assintoticamente eficiente sob a métrica de movimentações [Lipton and Naughton ].

Entre as suas principais características, destaca-se o fato de o *Cycle Sort* ser um algoritmo *in-place*, não estável e altamente sensível à presença de elementos duplicados, exigindo cuidados adicionais em sua implementação [Sedgewick and Wayne 2011]. Essa sensibilidade reforça a importância de análises cuidadosas do comportamento do algoritmo sob diferentes configurações de entrada, especialmente em vetores aleatórios ou com repetições de valores.

Por fim, o *Cycle Sort* apresenta um contraste relevante entre desempenho teórico e aplicabilidade prática, sendo pouco utilizado em sistemas gerais, mas altamente valioso em estudos comparativos e contextos específicos onde a minimização de escritas é prioritária. Essa particularidade justifica sua inclusão neste trabalho, permitindo avaliar como métricas distintas, além do tempo de execução, influenciam a escolha de algoritmos de ordenação. A figura 2 abaixo apresenta uma implementação em pseudocódigo do algoritmo.

**Listing 2. Implementação em Pseudocódigo do Cycle Sort.**

```
PROCEDIMENTO cycleSort(v[], n)
    PARA ciclo_inicio = 0 ATE n-2 FACA
        item = v[ciclo_inicio]
        pos = ciclo_inicio
```

```
        // Encontra posicao correta para o item
        PARA j = ciclo_inicio+1 ATE n-1 FACA
            SE v[j] < item ENTAO
                pos = pos + 1
            FIM SE
        FIM PARA

        // Se o item ja esta na posicao correta, continua
        SE pos != ciclo_inicio ENTAO
            // Trata elementos duplicados
            ENQUANTO item == v[pos] FACA
                pos = pos + 1
            FIM ENQUANTO

            // Troca o item com o elemento na posicao correta
            temp = v[pos]
            v[pos] = item
            item = temp

            // Continua o ciclo ate retornar a posicao inicial
            ENQUANTO pos != ciclo_inicio FACA
                pos = ciclo_inicio

                // Encontra nova posicao para o item
                PARA j = ciclo_inicio+1 ATE n-1 FACA
                    SE v[j] < item ENTAO
                        pos = pos + 1
                    FIM SE
                FIM PARA

                // Trata elementos duplicados novamente
                ENQUANTO item == v[pos] FACA
                    pos = pos + 1
                FIM ENQUANTO

                // Troca novamente
                temp = v[pos]
                v[pos] = item
                item = temp
            FIM ENQUANTO
        FIM SE
    FIM PARA
FIM PROCEDIMENTO
```

## 3. Metodologia

Nesta seção, são descritos o ambiente de execução empregado e os aspectos técnicos relacionados à implementação dos algoritmos, como bibliotecas utilizadas e características do hardware e do sistema operacional, de modo a assegurar a reprodutibilidade dos experimentos. Também são apresentados os mecanismos adotados para a coleta e o tratamento dos dados experimentais, bem como as ferramentas auxiliares empregadas na medição e análise de desempenho. Por fim, são explicitados os critérios e procedimentos experimentais utilizados para comparar o desempenho e o comportamento funcional dos algoritmos de ordenação sob diferentes configurações de entrada.

### 3.1. Especificações Técnicas

Esta subseção apresenta a configuração da máquina utilizada para a execução dos testes, fornecendo informações essenciais para a reprodutibilidade dos resultados. Essas informações permitem contextualizar os resultados obtidos, evidenciando as condições sob as quais os experimentos foram realizados. Observe a tabela 2.

**Tabela 2. Especificações Técnicas da Máquina Utilizada nos Testes**

| Especificação | Descrição |
|---|---|
| Processador | Intel Core i5-12450H 12º Gen (2.00 GHz) (8 núcleos, 12 threads, 12 MB cache) |
| Memória RAM | 16,0 GB @ 3200 MHz (utilizável: 15,7 GB) |
| SO | Windows 11 Home Single Language (Executado no Ubuntu 24.04.3 LTS via WSL2) |

## 3.2. Especificações Práticas

Na condução dos experimentos, foram desenvolvidas funções específicas em linguagem C para a geração dos vetores de entrada, contemplando distribuições crescentes, decrescentes e aleatórias, de modo a controlar a ordenação inicial dos dados. Essa padronização dos conjuntos de teste permitiu isolar o impacto da organização dos dados sobre o comportamento dos algoritmos, conectando a geração dos vetores às análises de desempenho realizadas posteriormente.

Os cenários experimentais consideraram vetores com 20.000, 40.000 e 60.000 elementos, sendo cada tamanho avaliado sob as três configurações de ordenação inicial definidas. Essa combinação sistemática de tamanhos e distribuições assegurou uma cobertura abrangente dos casos analisados, estabelecendo uma base consistente para a comparação entre *Shell Sort* e *Cycle Sort*.

Para a medição do tempo de execução, foi implementada uma função dedicada utilizando a chamada *clock_gettime* com o relógio *CLOCK_MONOTONIC*, garantindo maior precisão e independência de ajustes do sistema. Essa abordagem permitiu a obtenção do tempo decorrido em milissegundos, conectando a medição temporal às demais métricas coletadas durante a execução dos algoritmos.

Além do tempo de execução, foram captadas métricas adicionais, incluindo análise de complexidade observada, número de comparações, quantidade de trocas e consistência temporal, esta última calculada com base no desvio padrão das execuções. Algumas dessas métricas, como comparações e trocas, exigiram adaptações diretas no código dos algoritmos, permitindo relacionar o comportamento interno das rotinas de ordenação aos resultados experimentais obtidos. Observe a tabela 3.

**Tabela 3. Critérios de Avaliação e Justificativas**

| Critério de Avaliação | Justificativa para a Escolha |
|---|---|
| Tempo Médio de Execução (ms) | Permite avaliar o desempenho temporal dos algoritmos, sendo uma métrica fundamental para a comparação da eficiência prática sob diferentes tamanhos e configurações de entrada. |
| Número de Comparações | Reflete o custo lógico associado ao processo de ordenação, possibilitando relacionar o comportamento observado com a complexidade assintótica teórica baseada em comparações. |

| Critério de Avaliação | Justificativa para a Escolha |
|---|---|
| Quantidade de Trocas | Indica o volume de movimentações realizadas na memória, sendo especialmente relevante para algoritmos como o *Cycle Sort*, cujo objetivo principal é minimizar o número de escritas. |
| Complexidade Assintótica Observada | Permite analisar empiricamente a taxa de crescimento do custo computacional em função do tamanho da entrada, possibilitando a comparação entre teoria e prática. |
| Consistência do Tempo de Execução (desvio padrão) | Avalia a estabilidade do desempenho dos algoritmos entre diferentes execuções, indicando a previsibilidade e a robustez dos resultados experimentais. |

Cada cenário foi executado 11 vezes para ambos os algoritmos, sendo desconsiderada a primeira execução no cálculo das métricas, com o objetivo de mitigar efeitos de *warm-up*. Os resultados das 10 execuções válidas foram armazenados individualmente em arquivos de texto, posteriormente consolidados em arquivos CSV, os quais serviram de base para a geração dos gráficos por meio de um *script* em Python utilizando as bibliotecas *pandas* e *matplotlib.pyplot*.

## 4. Resultados e Discussão

Nesta seção, são apresentados e analisados os dados obtidos nos testes de desempenho dos algoritmos de ordenação *Shell Sort* e *Cycle Sort*, considerando diferentes cenários de ordenação em larga escala. Além disso, os dados serão discutidos à luz das características teóricas de cada algoritmo, buscando relacionar os resultados práticos com o comportamento esperado dos algoritmos implementados.

### 4.1. Tempo Médio de Execução

Os resultados na figura 3 mostram um padrão claro no *Shell Sort*: o melhor tempo ocorre no cenário crescente (já ordenado), seguido pelo decrescente, e o pior tempo no aleatório. Esta ordem é intrigante, pois o caso decrescente, tradicionalmente um pior caso teórico, apresenta desempenho intermediário. A explicação reside na eficiência da sequência de *gaps* utilizada. Para um vetor inversamente ordenado, a primeira passada com um *gap* largo reorganiza os dados de forma drástica, criando rapidamente uma pré-ordenação que as passadas subsequentes finalizam com eficiência.

O cenário aleatório se consolida como o mais custoso, configurando-se como o pior caso prático nesta implementação. A desordem completa não oferece padrões que possam ser otimizados pelas passadas do *Shell Sort*. Cada incremento na sequência de *gaps* precisa realizar uma quantidade significativa de comparações e movimentações para impor uma ordem, sem se beneficiar de qualquer estrutura inicial nos dados.

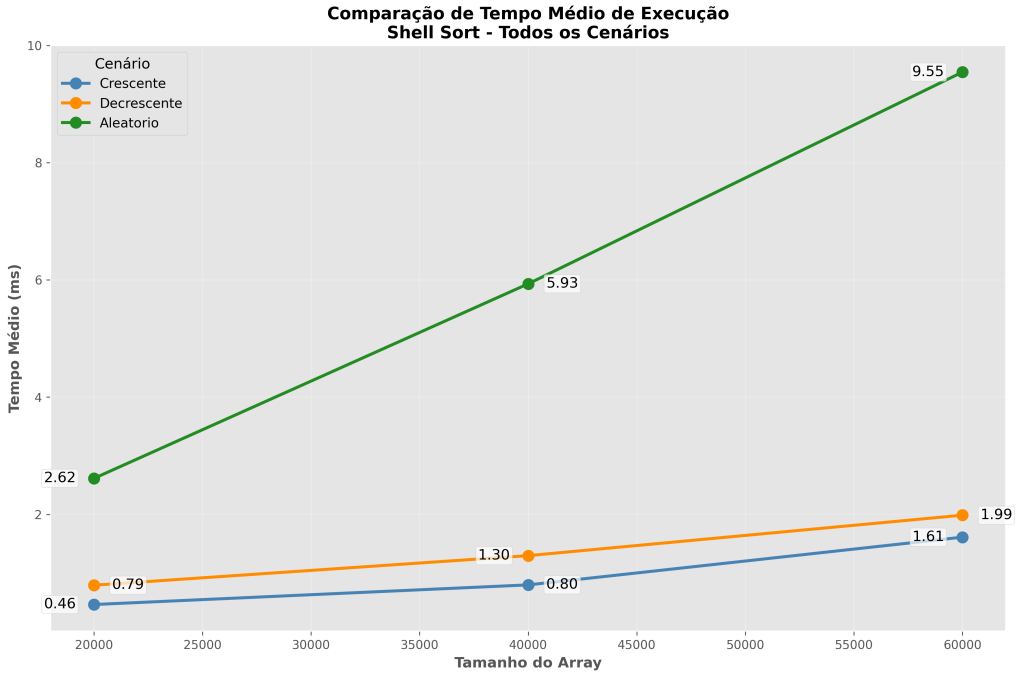


**Figura 3. Comparação de Tempo Médio de Execução do Shell Sort**

Portanto, a *performance* relativa dos cenários é determinada pela interação entre a estrutura inicial da entrada e a sequência de *gaps*. O algoritmo demonstra alta eficiência em dados com padrão (ordenado ou reverso) e maior custo em dados sem padrão, validando sua natureza e a importância da escolha da sequência de intervalos para seu desempenho.

Os dados presentes na figura 4 confirmam a natureza quadrática $O(n^2)$ do *Cycle Sort* em todos os cenários. Quando o tamanho do vetor triplica, os tempos aumentam por um fator entre 9,0 e 9,2. Este crescimento aproximadamente nônuplo é a assinatura empírica de uma complexidade quadrática, onde o custo é proporcional ao quadrado do tamanho da entrada.

A hierarquia de desempenho é mantida em todas as escalas. O cenário crescente permanece o mais rápido, pois evita a formação de ciclos longos. O decrescente apresenta tempo aproximadamente 1,9 vezes maior que o crescente para cada tamanho, um custo fixo adicional para desfazer a ordem inversa. O aleatório consolida-se como o pior caso, com tempos cerca de 3 vezes superiores aos do cenário decrescente na mesma escala.

Esta análise evidencia que a principal vantagem teórica do *Cycle Sort* – o número mínimo de escritas – é ofuscada por seu alto custo de comparações e busca. Seu uso é justificável apenas quando o custo de movimentação (sobrescrita) dos dados é exorbitante, e não para a otimização geral de tempo de execução em ordenação interna.

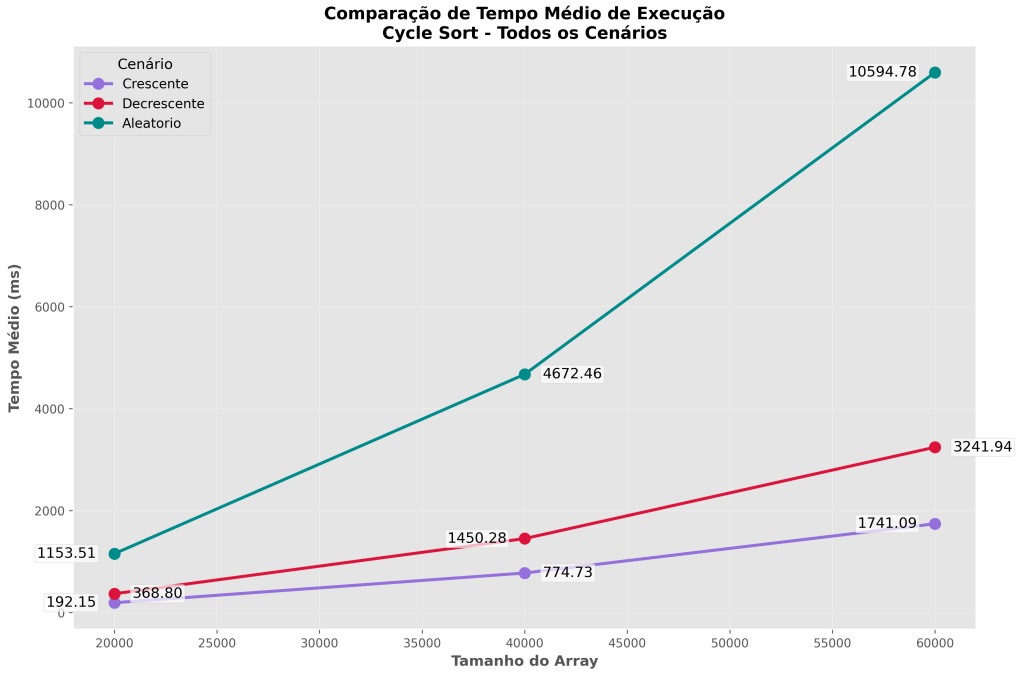


**Figura 4. Comparação de Tempo Médio de Execução do Cycle Sort**

## 4.2. Número de Comparações

Os dados de comparações do *Shell Sort* fornecidos pela figura 5 refletem a eficiência do algoritmo em explorar a ordenação pré-existente. No cenário crescente, o número de comparações é o mais baixo e cresce quase linearmente, indicando que o algoritmo realiza verificações mínimas ao confirmar a ordem já estabelecida. O cenário decrescente exige aproximadamente 1,6 vezes mais comparações que o crescente, evidenciando o custo de reorganizar inversões iniciais com a sequência de *gaps*.

O cenário aleatório demanda um número significativamente maior de comparações, cerca de 3,8 vezes superior ao crescente na maior entrada. Este salto comprova que a desordem completa força o algoritmo a realizar a ordenação integral, sem poder otimizar com base em *subarrays* parcialmente ordenados. A curva de crescimento é superlinear, coerente com a complexidade esperada.

A relação entre as contagens de comparações corrobora diretamente os tempos de execução medidos. A maior carga operacional no caso aleatório justifica seu desempenho mais lento, enquanto a eficiência nos casos ordenado e reverso valida a sensibilidade do *Shell Sort* à estrutura inicial dos dados.

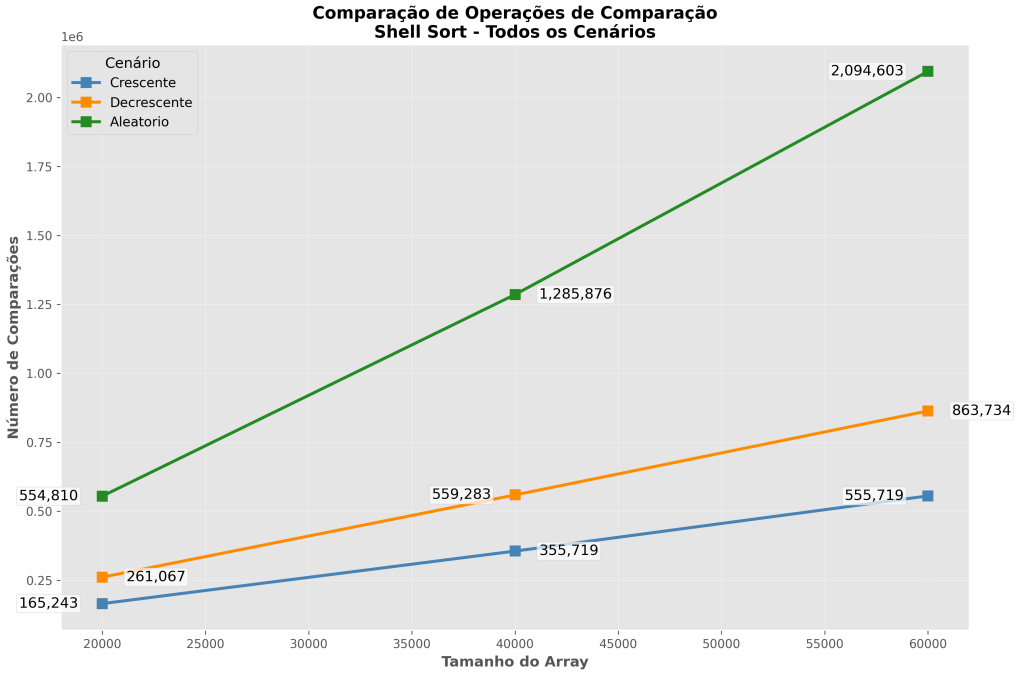


**Figura 5. Número de Comparações por Cenário no Shell Sort**

A escala das comparações no *Cycle Sort* mostrada na figura 6 é drasticamente superior à do *Shell Sort*, atingindo ordens de bilhões. Isso revela a ineficiência intrínseca do algoritmo em termos de comparações. No cenário crescente, as contagens seguem um crescimento próximo ao quadrático, confirmando a fórmula $n(n-1)/2$ para verificar uma lista já ordenada.

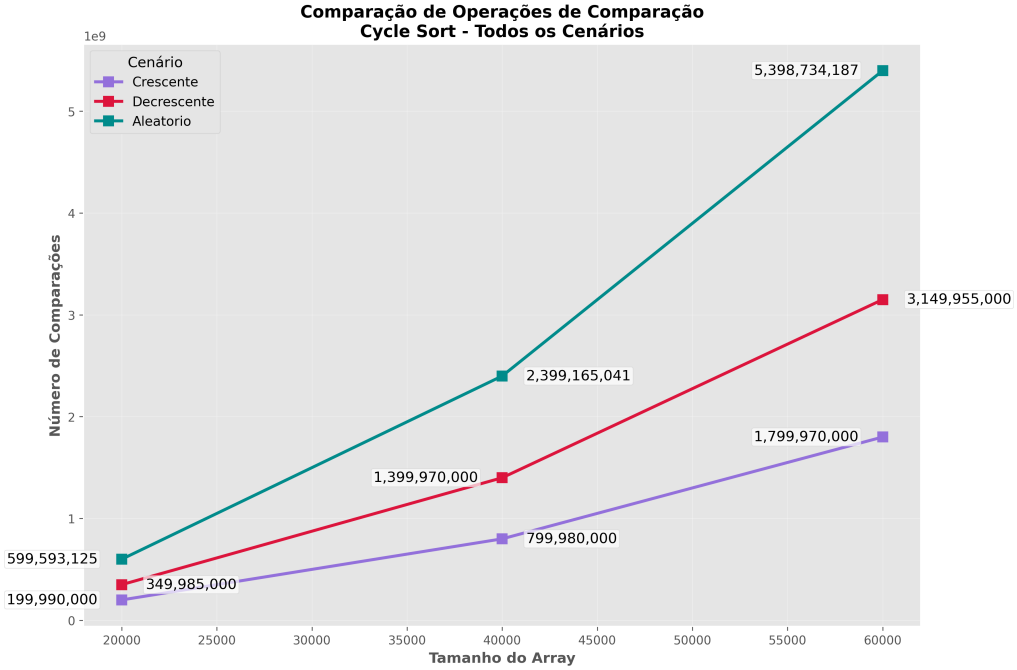


**Figura 6. Número de Comparações por Cenário no Cycle Sort**

O cenário decrescente exige aproximadamente 1,75 vezes mais comparações que o crescente para a mesma entrada. O caso aleatório consolida-se como o pior, exigindo cerca de 3 vezes mais comparações que o crescente. Esta enorme discrepância explica diretamente os tempos de execução observados: o custo de busca para posicionar cada elemento em um *array* sem padrão é proibitivo.

A análise comprova que a suposta vantagem do *Cycle Sort* em escritas é completamente anulada por seu custo quadrático em comparações. O algoritmo não é escalável para grandes conjuntos de dados, independentemente do cenário inicial, tornando-o uma escolha prática apenas para situações muito específicas onde a escrita é a operação criticamente dominante.

### 4.3. Quantidade de Trocas

Os dados de trocas do *Shell Sort* apresentados na figura 7 confirmam a eficiência do algoritmo. No cenário crescente, nenhuma troca é realizada, pois as inserções já encontram os elementos em posições corretas. Para o decrescente, o número de trocas cresce de forma superlinear, refletindo o esforço para corrigir inversões usando a sequência de *gaps*. Contudo, o volume é moderado, evidenciando a eficácia dos *gaps* em reduzir movimentações.

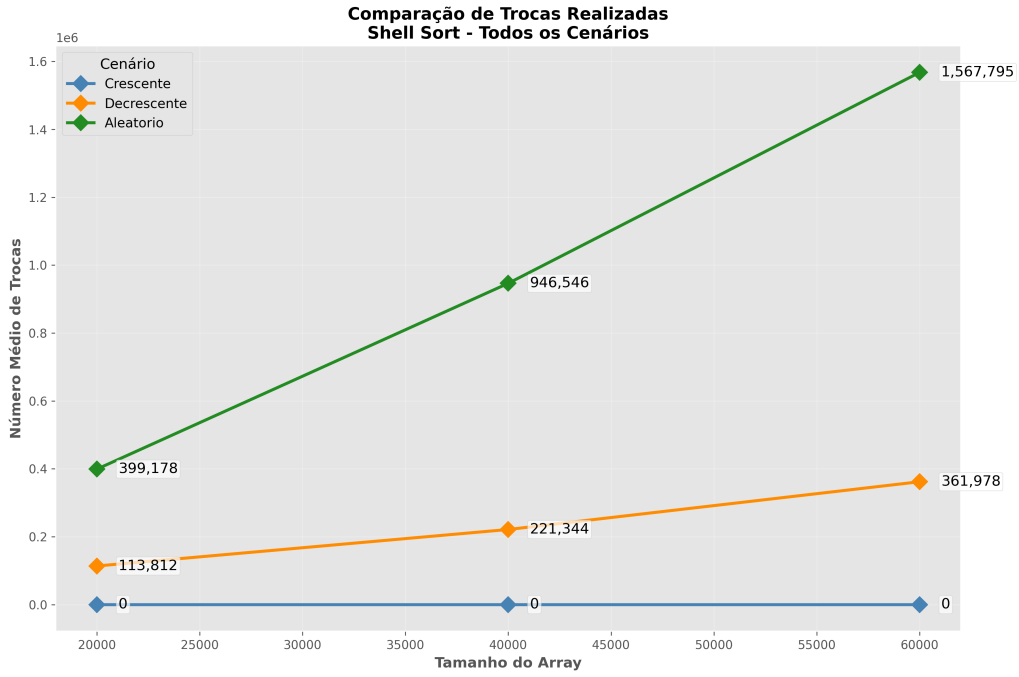


**Figura 7. Número de Trocas por Cenário no Shell Sort**

O cenário aleatório apresenta o maior número de trocas, com crescimento acentuado. Isto era esperado, dado que a ordenação precisa ser construída integralmente. A relação com as comparações é direta: mais comparações para localizar posições corretas resultam em mais trocas necessárias para reposicionar elementos fora de lugar.

Este padrão corrobora os resultados de tempo. O custo de movimentação, embora existente, não é o fator dominante no *Shell Sort*. A *performance* é mais impactada pelo número de comparações, o que explica sua superioridade sobre o *Cycle Sort*, que minimiza trocas, mas paga um preço exorbitante em comparações.

Os resultados observados na figura 8 comprovam a principal vantagem teórica do *Cycle Sort*: realizar o número mínimo de escritas necessárias. Nos cenários crescente e aleatório, o número de trocas é praticamente n-1 ou inferior, significando que cada elemento é movido no máximo uma vez para sua posição final. Esta é uma eficiência inigualável em operações de escrita.

Este resultado evidencia o *trade-off* fundamental do *Cycle Sort*. A otimização extrema em escritas é alcançada às custas de um número quadrático e proibitivo de comparações. Portanto, sua aplicação é justificada apenas em contextos onde a escrita na memória é uma operação extraordinariamente lenta ou desgastante, e nunca como uma solução de ordenação geral.

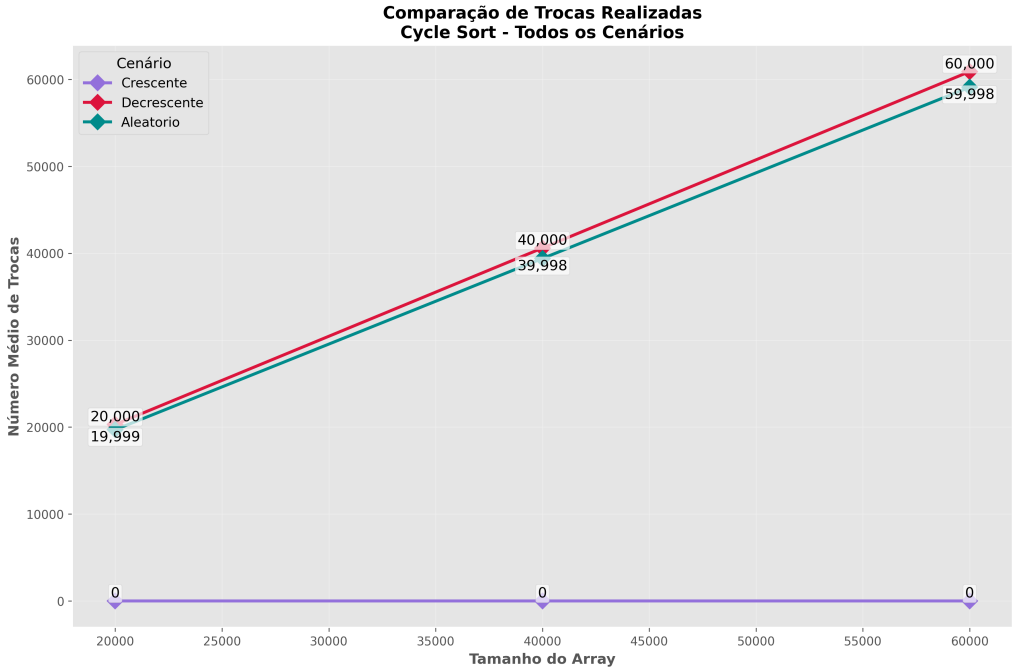


**Figura 8. Número de Trocas por Cenário no Cycle Sort**

No caso decrescente, o algoritmo atinge seu máximo teórico de trocas, que é n. Isto ocorre porque cada elemento está na posição oposta à sua final, forçando uma troca por item. Ainda assim, este é o mínimo absoluto para ordenar uma sequência totalmente invertida, uma eficiência notável.

### 4.4. Complexidade Assintótica Observada

No gráfico da figura 9, observa-se um comportamento significativamente diferente no *Shell Sort*. Os tempos médios são muito menores (ordem de milissegundos baixos) e crescem de forma bem mais suave com o aumento do tamanho do vetor. Os coeficientes $\alpha$ indicam um crescimento sublinear a quase linear, compatível com complexidades intermediárias entre $O(n)$ e $O(n \log n)$, dependendo do cenário e da sequência de *gaps* utilizada.

O cenário crescente apresenta o melhor desempenho, pois o *Shell Sort* se beneficia de dados parcialmente ordenados. Já o cenário decrescente, apesar de teoricamente

desfavorável para algoritmos simples como a variação do *Insertion Sort*, ainda mantém crescimento baixo. Por fim, o cenário aleatório é o mais custoso, mas permanece muito abaixo do *Cycle Sort* em termos absolutos. As curvas empíricas se mantêm próximas ou abaixo da referência $O(n \log n)$, distantes de $O(n^{1.5})$, sugerindo uma boa eficiência prática do algoritmo para os tamanhos testados.

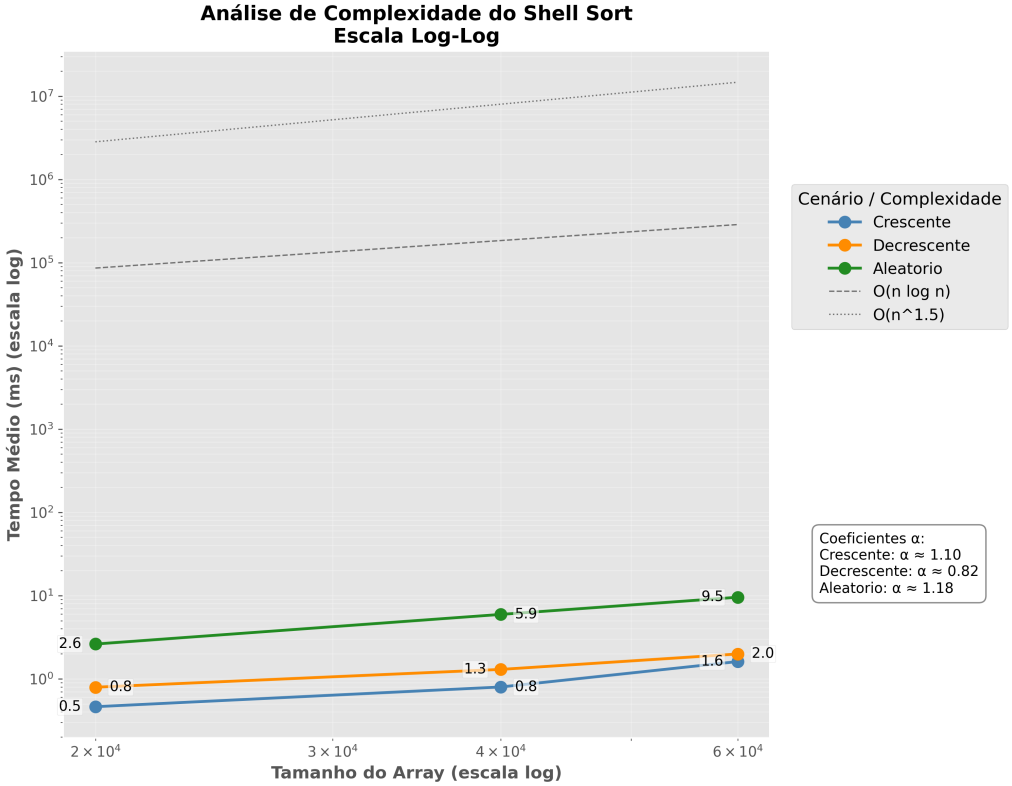


**Figura 9. Análise de Complexidade do Shell Sort**

O gráfico da figura 10 evidencia um crescimento acentuado do tempo de execução do *Cycle Sort* à medida que o tamanho do vetor aumenta. Em escala log-log, as curvas dos três cenários (crescente, decrescente e aleatório) apresentam inclinações muito semelhantes, o que é confirmado pelos coeficientes estimados.

Esses valores indicam claramente um comportamento quadrático, isto é, $O(n^2)$, independentemente da ordem inicial dos dados. Isso está de acordo com a natureza do *Cycle Sort*, que realiza, para cada posição, uma contagem do número de elementos menores no vetor inteiro, resultando em um número elevado de comparações.

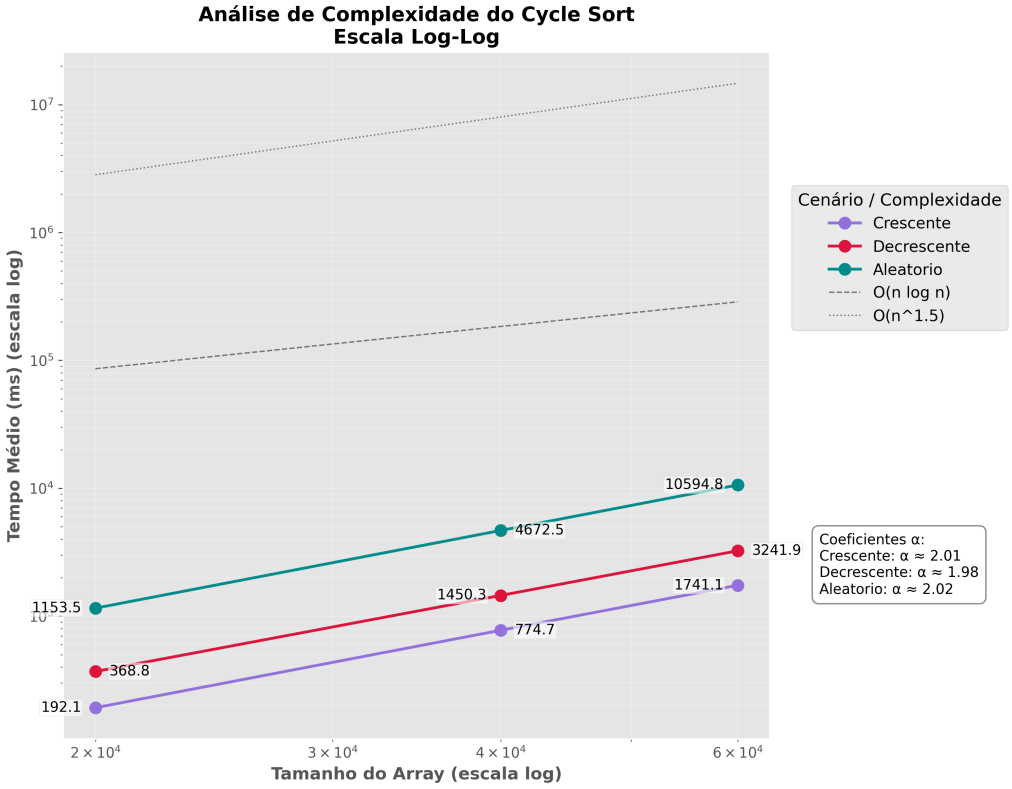


**Figura 10. Análise de Complexidade do Cycle Sort**

É importante ressaltar que o cenário aleatório apresenta os maiores tempos médios, seguido do decrescente e depois do crescente. E apesar de o *Cycle Sort* ser conhecido por minimizar o número de trocas, isso não se reflete em ganho de tempo, pois o custo dominante está nas comparações. As curvas empíricas ficam bem acima das referências teóricas $O(n \log n)$ e mais próximas de uma tendência $O(n^2)$, reforçando que o algoritmo não é adequado para grandes volumes de dados quando o critério é tempo de execução.

## 4.5. Consistência do Tempo de Execução

O gráfico do *Shell Sort* representado pela figura 11 revela uma variabilidade significativamente maior nos tempos de execução, especialmente nos cenários crescente e decrescente. Os coeficientes de variação alcançam valores superiores a 25% para o menor tamanho de entrada, indicando grande dispersão entre as execuções.

No cenário crescente, o coeficiente de variação apresenta comportamento não monotônico, com forte queda no tamanho intermediário e um aumento expressivo no maior tamanho analisado, sugerindo maior sensibilidade a fatores como cache, *branch prediction* e variações internas do algoritmo. O cenário decrescente mostra uma redução progressiva da variação conforme o tamanho do vetor aumenta, indicando maior estabilidade para entradas maiores. O cenário aleatório apresenta os menores coeficientes de variação entre os três cenários, mantendo-se relativamente estável e abaixo de 7%, o que sugere comportamento mais previsível quando os dados não possuem estrutura prévia.

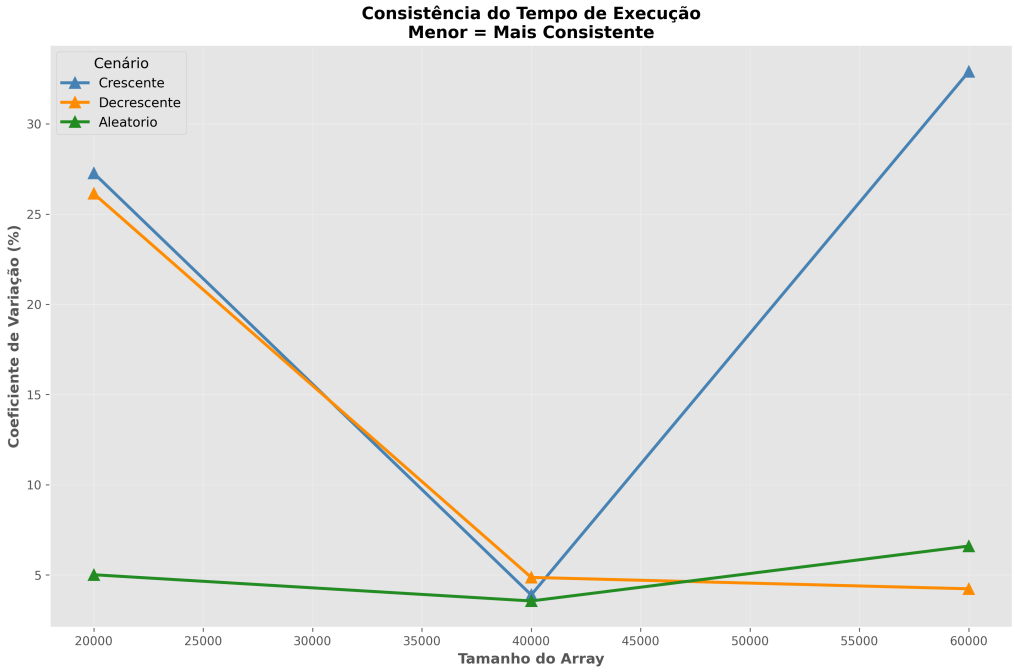


**Figura 11. Consistência Sobre o Tempo de Execução do Shell Sort**

O gráfico referente ao *Cycle Sort* na figura 12 indica que o algoritmo apresenta alta consistência no tempo de execução em todos os cenários analisados. Os coeficientes de variação permanecem inferiores a 2% para todos os tamanhos de entrada, o que evidencia uma baixa dispersão dos tempos medidos entre diferentes execuções.

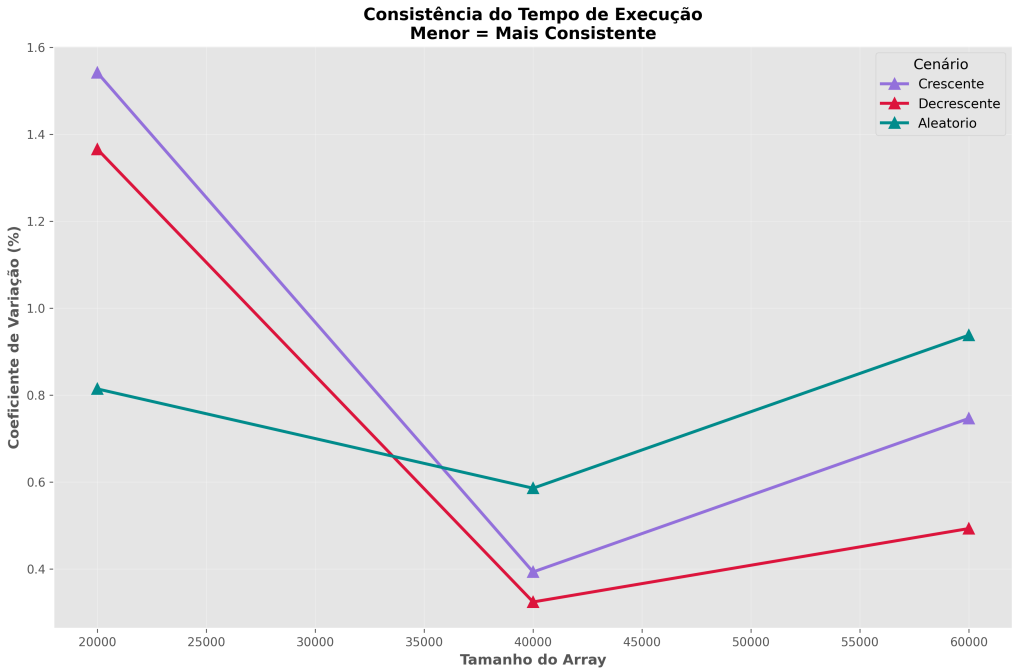


**Figura 12. Consistência Sobre o Tempo de Execução do Cycle Sort**

O cenário crescente apresenta maior variação para o menor tamanho de entrada,

porém ainda dentro de valores considerados muito baixos. À medida que o tamanho do vetor aumenta, há uma redução do coeficiente de variação, especialmente para o tamanho intermediário, indicando maior estabilidade do algoritmo conforme cresce o volume de dados. O cenário aleatório mantém valores relativamente estáveis ao longo dos tamanhos analisados, reforçando que a ordem inicial dos dados exerce pouca influência sobre a previsibilidade do tempo de execução.

Essa elevada consistência está diretamente relacionada à natureza do *Cycle Sort*, cujo fluxo de execução e número de comparações tendem a ser determinísticos e pouco sensíveis à disposição inicial dos elementos.

## 5. Conclusão

Os testes realizados confirmam as diferenças fundamentais entre os algoritmos *Shell Sort* e *Cycle Sort* em larga escala. O *Shell Sort* demonstrou desempenho prático superior, com complexidade próxima de $O(n \log n)$ e alta eficiência em todos os cenários testados. Sua *performance* foi sensível à ordenação inicial, sendo notavelmente rápida em dados pré-ordenados.

Por outro lado, o *Cycle Sort* exibiu o comportamento quadrático $O(n^2)$ esperado, tornando-se rapidamente inviável para grandes volumes de dados. Sua principal vantagem teórica — a minimização de escritas — não se traduziu em eficiência geral, pois o custo dominante revelou-se nas comparações. A consistência em seu tempo de execução não compensou sua lentidão absoluta.

A escolha entre os algoritmos deve, portanto, considerar o critério de desempenho prioritário. Para a ordenação geral com foco em velocidade, o *Shell Sort* é a alternativa claramente mais adequada. Já o *Cycle Sort* só se justifica em contextos extremamente específicos, onde a operação de escrita na memória é o fator crítico a ser otimizado, custo que seja o tempo total de processamento.

Por fim, os resultados validam a importância da análise empírica aliada à teoria. A escalabilidade do *Shell Sort* o consolida como uma ferramenta robusta para aplicações reais. A implementação do *Cycle Sort*, embora academicamente relevante, serve como demonstração prática dos *trade-offs* inerentes ao design de algoritmos.

## Referências


Ayazuddin, R. (2025). A comprehensive study of sorting algorithm performance using real-world dataset metrics.

Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265.

Donald, E. et al. (1997). The art of computer programming, volume 3: Sorting and searching.-2nd.

Frank, R. and Lazarus, R. (1960). A high-speed sorting procedure. *Communications of the ACM*, 3(1):20–22.

Gonnet, G. H. and Baeza-Yates, R. (1991). *Handbook of algorithms and data structures: in Pascal and C*. Addison-Wesley Longman Publishing Co., Inc.

Haddon, B. K. (1990). Cycle-sort: a linear sorting method. *The Computer Journal*, 33(4):365–367.

Hibbard, T. N. (1962). Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM (JACM)*, 9(1):13–28.

Hopcroft, J. E., Ullman, J. D., and Aho, A. V. (1983). *Data structures and algorithms*, volume 175. Addison-wesley Boston, MA, USA:.

Jain, R. (1990). *The art of computer systems performance analysis*. john wiley & sons.

Janson, S. and Knuth, D. E. (1997). Shellsort with three increments. *Random Structures & Algorithms*, 10(1-2):125–142.

Lipton, L. R. and Naughton, J. Knu d. knuth, the art of computer programming, vol. 3: Sorting and searching, addison-wesley, reading, ma, 1973. kri p. krishnan, online prediction algorithms for databases and operating systems,"brown univ.

Mahmoud, H. M. (2011). *Sorting: A distribution theory*. John Wiley & Sons.

Muhammad, M. H. G., Malik, J. A., Akhtar, M., Baloch, M. A., and Rajwana, M. A. (2025). Sort data faster: Comparing algorithms. *Southern Journal of Computer Science*, 1(02):28–37.

Pratt, V. R. (1972). Shellsort and sorting networks. Technical report.

Sedgewick, R. (1986). A new upper bound for shellsort. *Journal of Algorithms*, 7(2):159–173.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-wesley professional.

Shell, D. L. (1959). A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32.

Smythe, R. T. and Wellner, J. (2001). Stochastic analysis of shell sort. *Algorithmica*, 31(3):442–457.

Smythe, R. T. and Wellner, J. A. (2002). Asymptotic analysis of (3, 2, 1)-shell sort. *Random Structures & Algorithms*, 21(1):59–75.

Souza, R. M., Oliveira, F. S., and Pinto, P. E. (2018). Um limite superior para a complexidade do shellsort. In *Encontro de Teoria da Computação (ETC)*. SBC.

Sundaramoorthy, S. and Karunanidhi, G. (2025). A systematic analysis on performance and computational complexity of sorting algorithms. *Discover Computing*, 28(1):250.

Thomas H, C., Charles, E., Ronald L, R., Clifford, S., et al. (2009). Introduction to algorithms third edition.

Tokuda, N. (1992). An improved shellsort. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture-Information Processing'92, Volume 1-Volume I*, pages 449–457.

Turzo, N. A., Sarker, P., Kumar, B., Ghose, J., and Chakraborty, A. (2020). Defining a modified cycle sort algorithm and parallel critique with other sorting algorithms. *Global Research Development (GRD) ISSN: 2455-5703*, 5(4):1–8.