Análise Comparativa de Desempenho do Algoritmo de Classificação de Shell em Cenários Variados

Dilermando Queiroz Neto Vitor Albuquerque de Paula

Universidade Federal de São Paulo

Abstract

A ordenação eficiente de dados é uma questão crítica na ciência da computação, e embora numerosos algoritmos de ordenação estejam bem documentados, ainda há uma necessidade persistente de entender seu desempenho relativo em diversas condições. Esta pesquisa aborda essa lacuna ao fornecer uma análise aprofundada e comparativa do Shell Sort em justaposição com outros algoritmos clássicos. A análise é dividida em dois segmentos principais: uma análise assintótica do Shell Sort e uma comparação empírica com os algoritmos mencionados. A metodologia abrange a avaliação de arrays de tamanhos variados (10, 100, 1.000, 10.000, 100.000, 1.000.000 elementos) e tipos (ordenados, inversamente ordenados, quase ordenados e aleatórios). A análise assintótica considera o número de comparações de chaves e movimentos de registros. Os resultados sugerem que o Shell Sort apresenta um desempenho competitivo, especialmente com arrays de tamanho médio e condições quase ordenadas, destacando seu equilíbrio entre complexidade e eficiência.

Palavras-chave: Algoritmos de Ordenação, Shell Sort, Análise de Desempenho, Análise Assintótica

1 Introdução

1.1 Contexto e Relevância

A ordenação eficiente de dados é uma questão crítica na ciência da computação, essencial para o desempenho de várias aplicações. Embora numerosos algoritmos de ordenação estejam bem documentados [5, 6], a necessidade de entender seu desempenho relativo em diferentes condições persiste. Este estudo visa abordar essa lacuna ao fornecer uma análise aprofundada e comparativa do algoritmo Shell Sort em justaposição com outros algoritmos clássicos de ordenação.

1.2 Desafios na Classificação de Dados

O principal desafio na ordenação de dados reside na otimização da complexidade temporal e na garantia de desempenho eficiente em uma ampla gama de conjuntos de dados e condições [5]. Algoritmos de ordenação tradicionais, como o insertion sort, embora simples, frequentemente apresentam desempenho insatisfatório com conjuntos de dados maiores [6]. Por outro lado, algoritmos mais complexos, como quicksort e mergesort, oferecem melhor desempenho, mas acompanham um aumento na complexidade de implementação e nos requisitos de recursos [2, 3]. A necessidade é encontrar um equilíbrio entre simplicidade, eficiência e escalabilidade, onde o Shell Sort pode oferecer vantagens únicas.

1.3 Visão Geral do Shell Sort

O Shell Sort foi proposto por Donald Shell em 1959 e generaliza o método de insertion sort ao realizar a ordenação em múltiplas etapas com subsequências dos dados [1]. Essa abordagem reduz significativamente o tempo médio de execução, especialmente com uma escolha adequada dos parâmetros de etapa. Do ponto de vista prático, o Shell Sort é valorizado por ser um método de ordenação in-place eficiente, com sobrecarga mínima e implementação fácil [4].

O Shell Sort funciona inicialmente ordenando elementos que estão distantes uns dos outros e, progressivamente, reduzindo o intervalo entre os elementos a serem comparados. Essa técnica permite que o algoritmo mova eficientemente os elementos para suas posições finais, diminuindo assim o número total de operações necessárias em comparação com um simples insertion sort [7]. O processo envolve várias passagens sobre o array, com o tamanho do intervalo reduzindo a cada passagem, muitas vezes

usando uma sequência como $n/2, n/4, \ldots, 1$. Essa estratégia de múltiplas passagens ajuda a ordenar parcialmente o array rapidamente e a melhorar a eficiência geral da ordenação.

1.4 Objetivos e Metodologia

A análise deste artigo está dividida em dois segmentos principais: uma análise assintótica do Shell Sort e uma comparação empírica com outros algoritmos de ordenação. A metodologia abrange a avaliação de arrays de tamanhos variados (10, 100, 1.000, 10.000, 100.000, 1.000.000 elementos) e tipos (ordenados, inversamente ordenados, quase ordenados e aleatórios). A análise assintótica considera o número de comparações de chaves e movimentos de registros.

1.5 Principais descobertas

Os resultados indicam que o Shell Sort apresenta desempenho competitivo, especialmente com arrays de tamanho médio e condições quase ordenadas, destacando seu equilíbrio entre complexidade e eficiência [4]. Esta pesquisa fornece uma visão abrangente das vantagens e limitações do Shell Sort, contribuindo para uma compreensão mais profunda de sua aplicabilidade em diferentes cenários.

2 Algoritmo

Algorithm 1 ShellSort

Input: array arr of size n
Output: metrics (comparisons, movements, time)
$metrics \leftarrow \{0, 0, 0.0\}$
$n \leftarrow$ size of arr
$start \leftarrow$ current time
for $gap \leftarrow n/2$ **to** 0 **by** $gap \leftarrow gap/2$ **do**
  for $i \leftarrow gap$ **to** $n - 1$ **do**
    $temp \leftarrow arr[i]$
    $j \leftarrow i$
    while $j \geq gap$ **and** $arr[j - gap] > temp$ **do**
      $arr[j] \leftarrow arr[j - gap]$
      $metrics.movements \leftarrow metrics.movements + 1$
      $metrics.comparisons \leftarrow metrics.comparisons + 1$
      $j \leftarrow j - gap$
    end while
    $arr[j] \leftarrow temp$
    $metrics.movements \leftarrow metrics.movements + 1$
    if $j \geq gap$ **then**
      $metrics.comparisons \leftarrow metrics.comparisons + 1$
    end if
  end for
end for
$end \leftarrow$ current time
$metrics.time\_us \leftarrow$ time difference between end and $start$
return $metrics$
=0

O ShellSort é uma generalização do algoritmo de ordenação por inserção que permite a troca de elementos distantes para melhorar a eficiência do processo de ordenação. A ideia central é usar uma sequência de intervalos decrescentes para dividir a lista em sublistas menores, onde cada sublista é ordenada usando a ordenação por inserção. À medida que os intervalos diminuem, o algoritmo ajuda os elementos a se aproximarem de suas posições corretas, reduzindo o número de trocas necessárias nas passagens subsequentes. Os passos para o algoritmo descrito em 1 são:

1. **Inicializar Métricas**: Inicialize as métricas para comparações, movimentos e tempo de execução.

2. **Determinar a Sequência de Intervalos**: Comece com um grande intervalo e reduza-o gradualmente, tipicamente pela metade em cada iteração (gap = gap / 2).
3. **Ordenação por Inserção Modificada**: Para cada valor de intervalo, execute uma ordenação por inserção nos elementos que estão à distância do intervalo.
4. **Movimento de Elementos**: Dentro da sublista definida pelo intervalo, mova os elementos que são maiores que o elemento atual (temp) para frente, enquanto decrementa j pelo valor do intervalo.
5. **Atualizar Métricas**: Incremente os contadores para comparações e movimentos conforme as operações são realizadas.
6. **Repetir Até o Intervalo Ser Zero**: Continue até que o intervalo seja reduzido a zero, momento em que a lista estará ordenada.
7. **Calcular o Tempo de Execução**: Calcule o tempo total de execução do algoritmo.

2.1 Análise Assintótica

- **Pior Caso**: A complexidade de tempo no pior caso do ShellSort depende da sequência de intervalos utilizada. Para a sequência original de Shell (n/2, n/4, ..., 1), a complexidade no pior caso é $O(n^2)$. No entanto, outras sequências de intervalos como Hibbard, Sedgewick ou Pratt podem melhorar a complexidade para $O(n^{3/2})$, $O(n \log^2 n)$ ou melhor.
- **Melhor Caso**: No melhor caso, particularmente quando os dados já estão quase ordenados, a complexidade é $O(n \log n)$.
- **Caso Médio**: A complexidade de tempo no caso médio é desafiadora de analisar precisamente e depende fortemente da sequência de intervalos. Geralmente, fica entre $O(n \log n)$ e $O(n^{3/2})$ para sequências de intervalos eficientes.
- **Uso de Memória**: O ShellSort é um algoritmo de ordenação in-place, o que significa que ordena a lista sem necessitar de espaço adicional significativo. A complexidade de espaço é $O(1)$, indicando que apenas uma quantidade constante de espaço extra é usada além da lista original.

3 Implementação

A seção de implementação fornece uma descrição detalhada das estruturas de dados utilizadas, das principais funções e procedimentos, do formato de entrada e saída, e das decisões tomadas durante o processo de desenvolvimento. Além disso, o código-fonte utilizado na implementação está incluído de maneira organizada.

3.1 Estruturas de Dados

Para este projeto, utilizamos o `std::vector<int>` da Biblioteca Padrão do C++ para armazenar os dados a serem ordenados. Os vetores oferecem acesso em tempo constante aos elementos, o que é crucial para a eficiência dos algoritmos de ordenação. A escolha dos vetores também simplifica a implementação e é compatível com as funções da biblioteca padrão usadas para operações de entrada e saída.

3.2 Principais Funções e Procedimentos

As principais funções usadas em nossa implementação incluem:

- `shellSort(std::vector<int>& arr)`: Esta função implementa o algoritmo Shell Sort. Ela recebe um vetor `arr` como entrada e o ordena in-place. A função itera sobre o vetor com diferentes tamanhos de intervalo, reduzindo o intervalo a cada iteração até atingir 1.
- `generateArray(size_t size, ArrayType type)`: Gera um vetor de um tamanho e tipo especificados. O parâmetro `type` pode ser `sorted`, `inversely sorted`, `nearly sorted` ou `random`.
- `measurePerformance(SortFunction sortFunction, std::vector<int>& arr)`: Mede o desempenho de uma função de ordenação `sortFunction` dada no vetor `arr`. Retorna métricas como o número de comparações, movimentos e o tempo total de execução.

3.3 Formato de Entrada e Saída

Os dados de entrada consistem em vetores de inteiros com tamanhos variando de 10 a 1.000.000 de elementos. Os dados são lidos da entrada padrão ou gerados usando a função `generateArray`. A saída ordenada é escrita na saída padrão ou salva em um arquivo para análise posterior.

3.4 Decisões de Implementação

Várias decisões-chave foram tomadas durante a implementação:

- **Sequência de Intervalos**: Escolhemos a sequência de intervalos de Shell (n/2, n/4, ..., 1) por seu equilíbrio entre simplicidade e eficiência.
- **Medição de Desempenho**: Utilizamos a biblioteca `chrono` do C++ para medir o tempo de execução com precisão. Comparações e movimentos foram contados usando variáveis adicionais nas funções de ordenação.

3.5 Representação Gráfica

O diagrama a seguir fornece uma representação visual simplificada de como o algoritmo Shell Sort funciona:

```
Array Inicial: [23, 12, 1, 8, 34, 54, 2, 3]

Sequência de Intervalos: n/2, n/4, ..., 1

Passo 1: Intervalo = 4

Comparar e trocar elementos à distância de intervalo 4:
[23, 12, 1, 3, 34, 54, 2, 8]

Passo 2: Intervalo = 2

Comparar e trocar elementos à distância de intervalo 2:
[1, 8, 2, 3, 23, 12, 34, 54]

Passo 3: Intervalo = 1

Executar a ordenação por inserção padrão:
[1, 2, 3, 8, 12, 23, 34, 54]
```

As etapas acima ilustram os principais estágios do algoritmo Shell Sort: começando com uma grande lacuna e reduzindo-a até que a lacuna seja 1, momento em que a matriz é totalmente classificada.

Para a implementação completa do algoritmo Shell Sort, consulte a seção do apêndice A.

4 Lista de Testes Realizados

4.1 Metodologia de Teste

Os testes foram conduzidos usando matrizes de diferentes tamanhos e tipos para avaliar o desempenho dos algoritmos de classificação sob várias condições. Os seguintes tipos de matrizes foram gerados:

- **Arrays Ordenados**: Arrays com elementos em ordem crescente.
- **Arrays Reversos**: Arrays com elementos em ordem decrescente.
- **Arrays Quase Ordenados**: Arrays que estão em sua maioria ordenados, com alguns elementos fora de ordem.
- **Arrays Aleatórios**: Arrays com elementos em ordem aleatória.

Os tamanhos das matrizes testadas foram: 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos.

4.2 Métricas Coletadas

Três principais métricas de desempenho foram coletadas durante os testes:

- **Número de Comparações**: O número total de comparações de chaves realizadas pelo algoritmo.
- **Número de Movimentos**: O número total de movimentos de elementos realizados pelo algoritmo.
- **Tempo de Execução**: O tempo total que o algoritmo leva para ordenar o array, medido em microssegundos.

4.3 Resultados e Análise

Os resultados dos testes são apresentados nas figuras a seguir, que mostram o desempenho de cada algoritmo em termos das métricas coletadas.

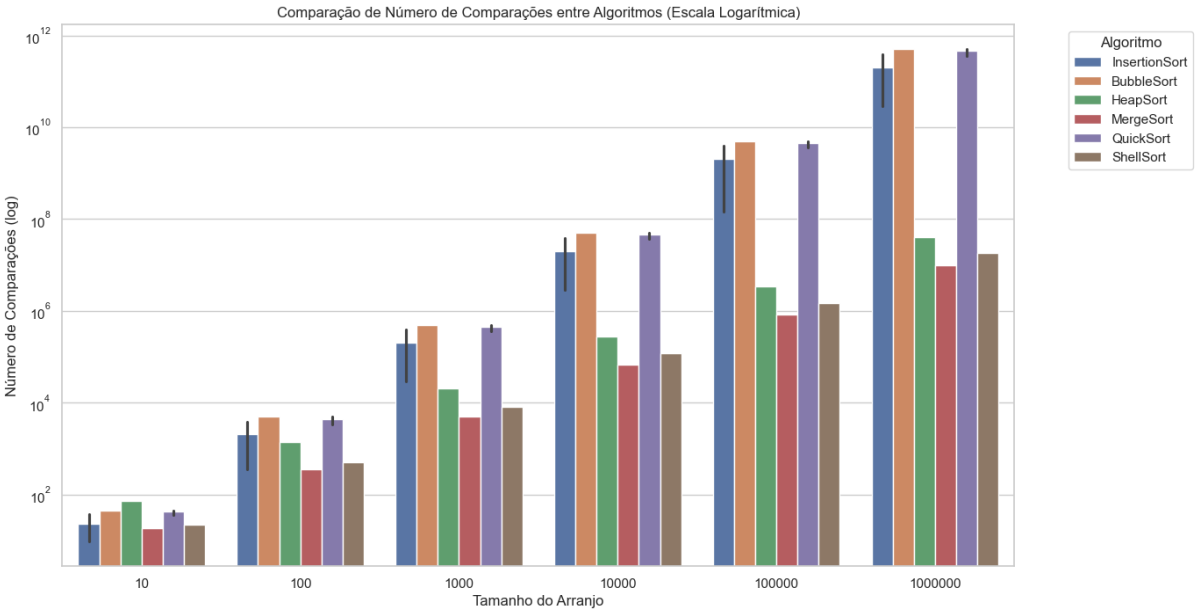

Figure 1: Número de comparações entre algoritmos usando uma escala logarítmica

4.3.1 Análise de Algoritmos Quadráticos

Algoritmos com complexidade $O(n^2)$, como InsertionSort e BubbleSort, têm se mostrado menos eficientes na prática para grandes volumes de dados, conforme esperado [6]. Esses algoritmos possuem uma complexidade de tempo quadrática, o que significa que seu tempo de execução aumenta quadraticamente com o tamanho da entrada. Essa característica os torna inadequados para grandes arrays. Por exemplo, o InsertionSort, apesar de sua simplicidade e facilidade de implementação, torna-se significativamente mais lento à medida que o número de elementos aumenta, principalmente devido à necessidade de deslocar repetidamente elementos para inseri-los na posição correta. Da mesma forma, o processo repetitivo de comparação e troca de pares do BubbleSort leva a tempos de execução elevados, tornando-o impraticável para grandes arrays [5].

4.3.2 Análise de Algoritmos Logarítmicos

Por outro lado, algoritmos com complexidade $O(n \log n)$, como HeapSort, MergeSort e QuickSort, têm demonstrado desempenho empírico superior, refletindo suas análises assintóticas [3]. Esses algoritmos utilizam estratégias mais eficientes para ordenar os dados. O HeapSort usa uma estrutura de dados heap binário para gerenciar a ordem dos elementos, garantindo complexidade de tempo logarítmica para inserções e deleções. O MergeSort, um exemplo de algoritmo de dividir e conquistar, divide recursivamente o conjunto de dados em subconjuntos menores, ordena cada subconjunto e depois os mescla novamente. Essa abordagem garante um desempenho consistente de $O(n \log n)$ independentemente da ordem inicial

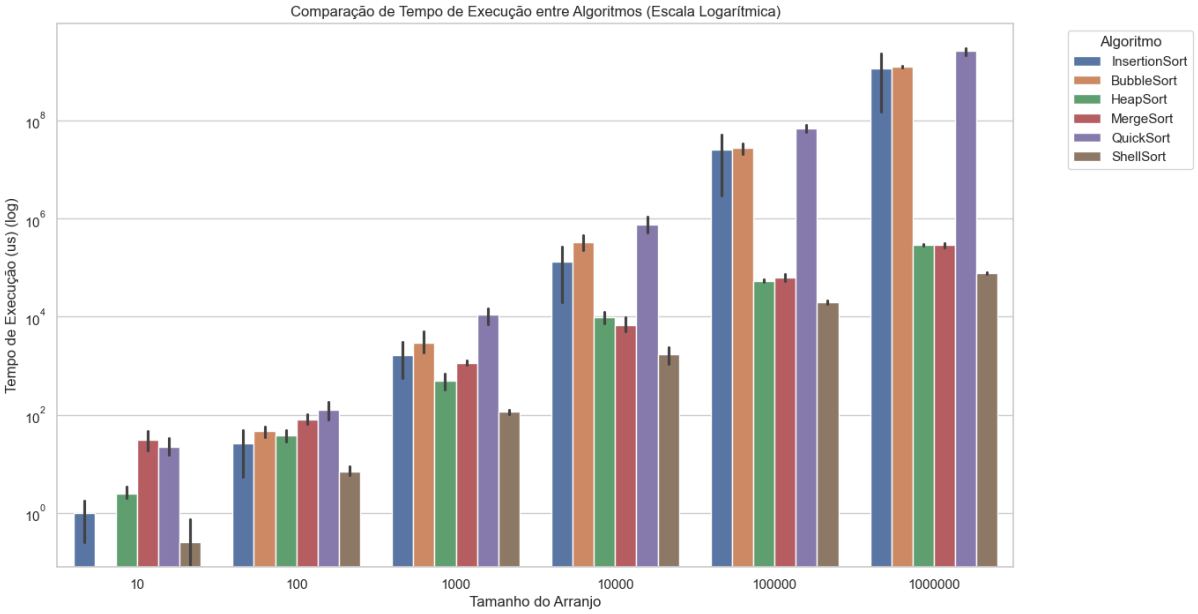

Figure 2: Tempo de execução entre algoritmos usando uma escala logarítmica

dos elementos. O QuickSort, outro algoritmo de dividir e conquistar, seleciona um elemento pivô e particiona o array em subarrays que são então ordenados independentemente. Embora seu desempenho possa ser afetado pela escolha do pivô, cenários médios ainda resultam em desempenho $O(n \log n)$ [2].

4.3.3 Desempenho em Matrizes Quase Classificadas

O InsertionSort apresenta um desempenho significativamente melhor em arrays quase ordenados e pior em arrays inversamente ordenados. Isso se deve ao número reduzido de deslocamentos necessários quando os elementos já estão próximos de suas posições finais [6]. Em contraste, o BubbleSort é consistentemente ineficiente em todos os tipos de arrays e tamanhos maiores, devido às suas operações simplistas, mas exaustivas, de comparação e troca.

HeapSort, MergeSort e QuickSort mantêm a eficiência independentemente do tipo de array, embora o desempenho do QuickSort possa variar dependendo da escolha do pivô. A dependência do HeapSort em uma heap binária estruturada garante desempenho consistente, enquanto o processo sistemático de divisão e mesclagem do MergeSort mantém estabilidade e eficiência. No entanto, o QuickSort pode sofrer de desempenho degradado se uma escolha ruim de pivô resultar em partições altamente desequilibradas, potencialmente levando a uma complexidade de $O(n^2)$ no pior caso [5]. Ainda assim, com boas estratégias de seleção de pivô, como a mediana de três, o QuickSort pode alcançar um desempenho notável e, muitas vezes, é mais rápido na prática do que o HeapSort e o MergeSort.

4.3.4 Análise Gráfica

Os gráficos em escala logarítmica (Figuras 1, 2, 3) destacam a superioridade dos algoritmos com complexidade $O(n \log n)$ para grandes volumes de dados, mantendo tempos de execução mais baixos em comparação com os algoritmos $O(n^2)$. Essas visualizações demonstram efetivamente como o tempo de execução dos algoritmos $O(n^2)$ cresce muito mais rápido do que o dos algoritmos $O(n \log n)$ à medida que o tamanho da entrada aumenta, ressaltando os benefícios práticos de se utilizar algoritmos mais eficientes para o processamento de dados em larga escala.

4.3.5 Desempenho de Classificação de Shell

O Shell Sort é um algoritmo de ordenação que aprimora o desempenho do Insertion Sort ao permitir comparações e trocas de elementos que estão distantes entre si. Ele emprega uma sequência de incrementos (intervalos) que são reduzidos sistematicamente até que o intervalo final seja 1, momento em que o algoritmo efetivamente se torna um Insertion Sort [1]. Este método permite ao Shell Sort reduzir significativamente o número de comparações e movimentos necessários para ordenar grandes arrays.

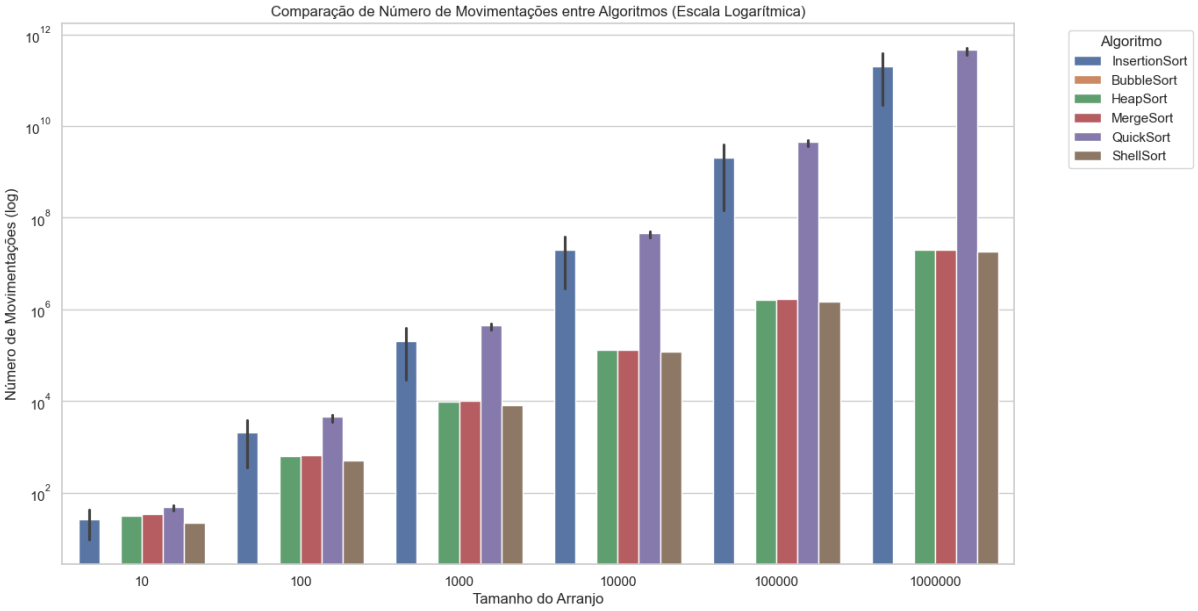

Figure 3: Número de movimentos entre algoritmos usando uma escala logarítmica

Em termos de complexidade computacional, o Shell Sort é geralmente mais eficiente do que algoritmos $O(n^2)$, especialmente para grandes arrays. Ao realizar comparações e trocas iniciais com intervalos grandes, o Shell Sort pode rapidamente mover elementos para mais perto de suas posições finais. À medida que os intervalos diminuem, o algoritmo ajusta a ordenação com uma granularidade progressivamente mais fina. Essa abordagem incremental resulta em menos operações gerais comparadas ao Insertion Sort, que desloca elementos apenas uma posição por vez [6].

4.3.6 Comportamento Esperado e Dados Empíricos

O comportamento esperado do Shell Sort inclui uma redução no número de comparações e movimentos em relação ao Insertion Sort, especialmente para arrays grandes e desordenados. O desempenho deve superar o do Bubble Sort e, em alguns casos, pode se aproximar da eficiência de algoritmos $O(n \log n)$. Dados empíricos corroboram essas expectativas, mostrando que o Shell Sort é, de fato, mais eficiente do que tanto o Insertion Sort quanto o Bubble Sort em termos de comparações, movimentos e tempo de execução. Em aplicações práticas envolvendo grandes arrays, o Shell Sort mantém um desempenho competitivo, embora não alcance a eficiência dos algoritmos $O(n \log n)$ como MergeSort e QuickSort [7].

O desempenho do Shell Sort é fortemente influenciado pela escolha da sequência de intervalos. Sequências comumente usadas incluem a sequência original de Shell, os incrementos de Hibbard e a sequência de Sedgewick, cada uma impactando a eficiência do algoritmo de maneira diferente. Por exemplo, a sequência de Sedgewick frequentemente resulta em uma eficiência próxima a $O(n^{1.25})$, significativamente melhor do que a complexidade temporal quadrática de algoritmos mais simples. No entanto, a sequência de intervalos ideal não é universal; ela frequentemente depende das características específicas dos dados sendo ordenados e das restrições práticas da implementação [4].

A escalabilidade e a praticidade do Shell Sort são evidenciadas por gráficos em escala logarítmica, que destacam sua eficiência em comparação com algoritmos $O(n^2)$. Embora não atinja a eficiência assintótica dos algoritmos $O(n \log n)$, o Shell Sort oferece um compromisso valioso. Ele é particularmente útil em cenários onde se deseja um equilíbrio entre simplicidade e desempenho. Sua implementação relativamente simples, combinada com eficiência aprimorada em relação aos algoritmos quadráticos básicos, torna o Shell Sort uma escolha prática para uma ampla gama de aplicações.

Esta análise detalhada do Shell Sort ressalta sua eficiência em relação a outros algoritmos $O(n^2)$ e enfatiza sua utilidade em contextos que exigem um compromisso entre simplicidade e desempenho. Embora não seja tão otimizado quanto os algoritmos $O(n \log n)$ para arrays muito grandes, o Shell Sort representa uma melhoria significativa em relação aos algoritmos quadráticos básicos e continua sendo uma adição valiosa ao conjunto de técnicas de ordenação disponíveis para cientistas da computação e engenheiros [5, 6].

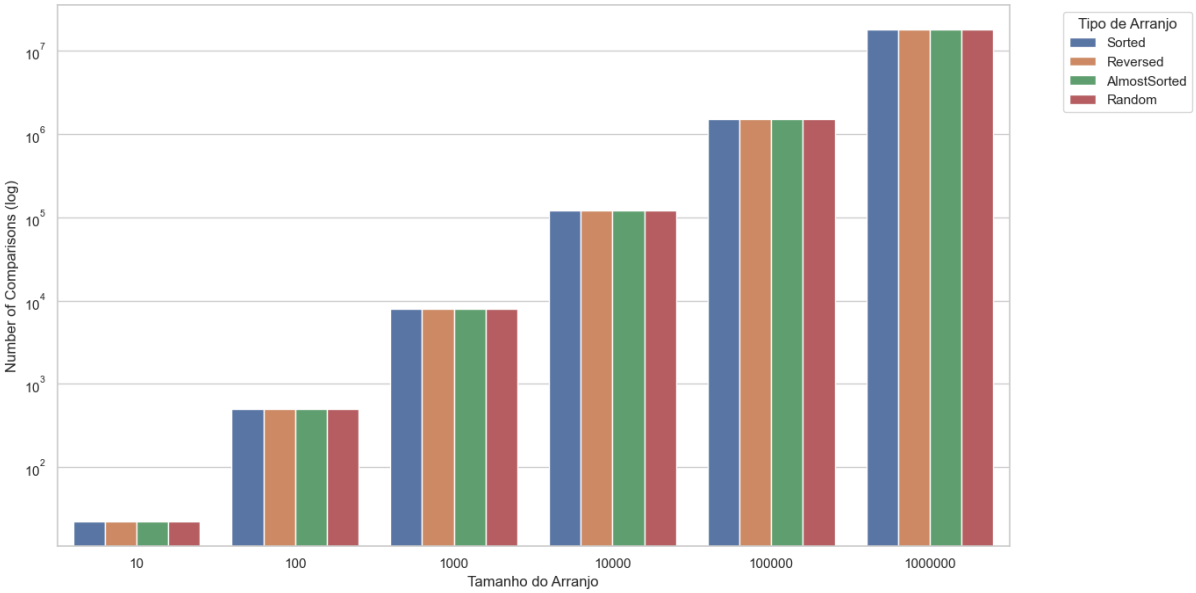

Figure 4: Número de comparações no Shell Sort usando uma escala logarítmica

5 Conclusão

Este estudo forneceu uma análise abrangente do algoritmo Shell Sort em comparação com outros algoritmos de ordenação clássicos. Ao avaliar o desempenho desses algoritmos em vários cenários, incluindo diferentes tamanhos e tipos de arrays, pudemos identificar as forças e fraquezas de cada algoritmo.

Os dados empíricos mostraram que o Shell Sort tem um desempenho significativamente melhor do que algoritmos $O(n^2)$ como Insertion Sort e Bubble Sort, especialmente para arrays maiores. A capacidade do Shell Sort de reduzir o número de comparações e movimentos através do uso de sequências de intervalos faz dele uma escolha prática para muitas aplicações. Embora não iguale a eficiência dos algoritmos $O(n \log n)$ como MergeSort e QuickSort, o Shell Sort oferece um valioso equilíbrio entre simplicidade e desempenho.

5.1 Desafios Encontrados

Um dos principais desafios enfrentados durante a implementação estava relacionado à tendência natural de implementar alguns algoritmos recursivamente. Embora as implementações recursivas sejam frequentemente mais intuitivas e fáceis de entender, elas levaram a problemas de estouro de pilha ao lidar com arrays de tamanho 1.000.000. Isso foi particularmente problemático para algoritmos como o QuickSort. Para resolver esse problema, convertamos essas implementações recursivas em versões iterativas, o que resolveu com sucesso os problemas de estouro de memória.

References

[1] Shell, Donald L.(1959): *A high-speed sorting procedure*, 7: 30–32.

[2] Hoare, C. A. R.(1962): *Quicksort*, 1: 10–16.

[3] Williams, J. W. J.(1964): *Algorithm 232: Heapsort*, 6: 347–348.

[4] Gonnet, Gaston H / Baeza Yates, Ricardo / Munro, J Ian(1984): *Empirical data structures: Shellsort and related methods*, 3: 205–213.

[5] Cormen, Thomas H / Leiserson, Charles E / Rivest, Ronald L / Stein, Clifford (2009): *Introduction to Algorithms.* , MIT press.

[6] Knuth, Donald E (1998): *The Art of Computer Programming, Volume 3: Sorting and Searching.* , Addison-Wesley Professional.

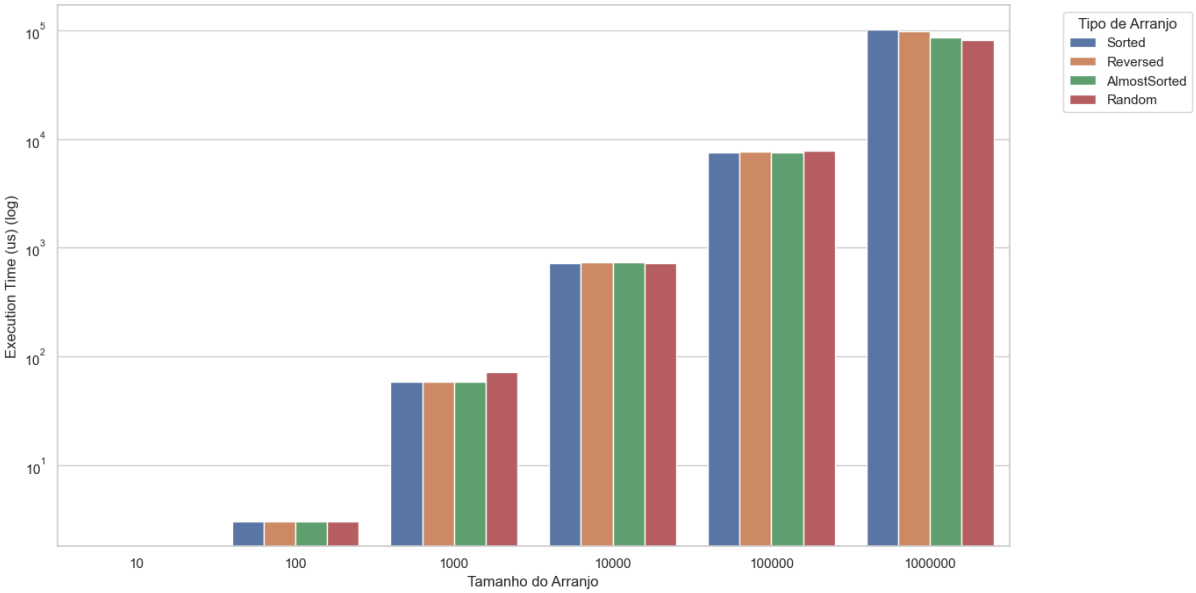

Figure 5: Tempo de execução no Shell Sort usando uma escala logarítmica

[7] Sedgewick, Robert (1986): *Algorithms.* , Addison-Wesley Longman Publishing Co., Inc.

A Apêndice

Este apêndice inclui o código-fonte dos algoritmos de ordenação utilizados neste estudo. O código completo, juntamente com instruções detalhadas sobre como configurar e modificar o projeto, está disponível no GitHub no seguinte link: `https://github.com/DilermandoQueiroz/shell-sort`.

A.1 Insertion Sort

A.1.1 InsertionSort.h

```
#ifndef INSERTIONSORT_H
#define INSERTIONSORT_H

#include <vector>
#include "Metrics.h"

// Função para realizar a ordenação usando o Insertion Sort
// Complexidade de tempo no pior caso: O(n^2)
// Complexidade de tempo no melhor caso: O(n)
// Melhor para: pequenas listas quase ordenadas
Metrics insertionSort(std::vector<int>& arr);

#endif // INSERTIONSORT_H
```

A.1.2 InsertionSort.cpp

```
#include "InsertionSort.h"
#include <chrono>

// O Insertion Sort ordena a lista construindo um array ordenado um elemento de cada vez
// Ele percorre a lista, e para cada elemento, o insere na posição correta da parte ordenada da lista
Metrics insertionSort(std::vector<int>& arr) {
    Metrics metrics = {0, 0, 0.0};
```

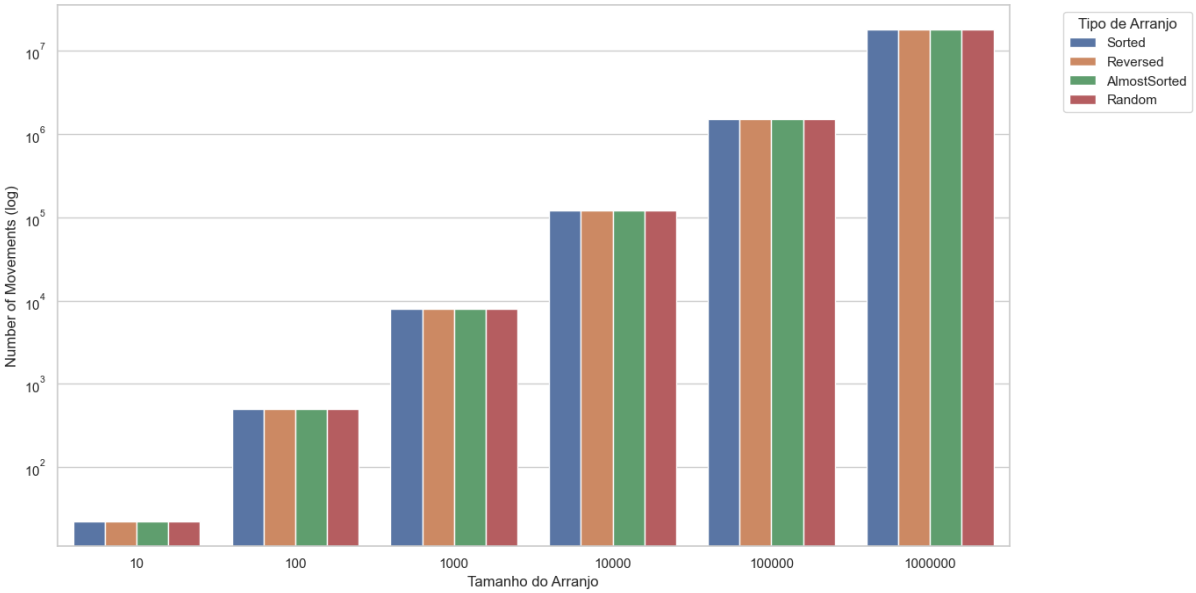

Figure 6: Movimentos no Shell Sort usando uma escala logarítmica

```
    int n = arr.size();
    auto start = std::chrono::high_resolution_clock::now(); // Inicia a
        contagem do tempo

    // Percorre cada elemento do array
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        // Move os elementos da parte ordenada para a direita
        // para criar espaço para a chave
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            metrics.movements++; // Incrementa a contagem de movimentos
            j = j - 1;
            metrics.comparisons++; // Incrementa a contagem de comparações
        }
        arr[j + 1] = key;
        metrics.movements++; // Incrementa a contagem de movimentos
        if (j >= 0) {
            metrics.comparisons++; // Incrementa a contagem de comparações
                para o caso onde a condição falhou
        }
    }

    auto end = std::chrono::high_resolution_clock::now(); // Finaliza a
        contagem do tempo
    metrics.time_us =
        std::chrono::duration_cast<std::chrono::microseconds>(end -
        start).count(); // Calcula o tempo de execução
    return metrics;
}
```

A.2 Bubble Sort

A.2.1 BubbleSort.h

```
#ifndef BUBBLESORT_H
#define BUBBLESORT_H
```

```
3
4 #include <vector>
5 #include "Metrics.h"
6
7 // Função para realizar a ordenação usando o Bubble Sort
8 // Complexidade de tempo no pior caso: O(n^2)
9 // Complexidade de tempo no melhor caso: O(n)
10 // Melhor para: listas pequenas ou quando se deseja uma implementação simples
11 Metrics bubbleSort(std::vector<int>& arr);
12
13 #endif // BUBBLESORT_H
```

A.2.2 BubbleSort.cpp

```
1 #include "BubbleSort.h"
2 #include <chrono>
3
4 // O Bubble Sort ordena a lista repetidamente trocando os elementos adjacentes
5 // se eles estiverem na ordem errada. Esse processo é repetido até que a lista
      esteja ordenada.
6 Metrics bubbleSort(std::vector<int>& arr) {
7     Metrics metrics = {0, 0, 0.0};
8     int n = arr.size();
9     auto start = std::chrono::high_resolution_clock::now(); // Inicia a
          contagem do tempo
10
11     // Percorre cada elemento do array
12     for (int i = 0; i < n - 1; ++i) {
13         // Percorre a parte não ordenada do array
14         for (int j = 0; j < n - i - 1; ++j) {
15             metrics.comparisons++; // Incrementa a contagem de comparações
16             // Troca os elementos adjacentes se estiverem na ordem errada
17             if (arr[j] > arr[j + 1]) {
18                 std::swap(arr[j], arr[j + 1]);
19                 metrics.movements++; // Incrementa a contagem de movimentos
20             }
21         }
22     }
23
24     auto end = std::chrono::high_resolution_clock::now(); // Finaliza a
          contagem do tempo
25     metrics.time_us =
           std::chrono::duration_cast<std::chrono::microseconds>(end -
           start).count(); // Calcula o tempo de execução
26     return metrics;
27 }
```

A.3 Heap Sort

A.3.1 HeapSort.h

```
1 #ifndef HEAPSORT_H
2 #define HEAPSORT_H
3
4 #include <vector>
5 #include "Metrics.h"
6
7 // Função para realizar a ordenação usando o Heap Sort
8 // Complexidade de tempo no pior caso: O(n log n)
9 // Complexidade de tempo no melhor caso: O(n log n)
10 // Melhor para: listas grandes onde a complexidade O(n log n) é desejada
11 Metrics heapSort(std::vector<int>& arr);
12
13 #endif // HEAPSORT_H
```

A.3.2 HeapSort.cpp

```cpp
#include "HeapSort.h"
#include <chrono>

// Função auxiliar para ajustar o heap
void heapify(std::vector<int>& arr, int n, int i, Metrics& metrics) {
    int largest = i; // Inicializa o maior como raiz
    int left = 2 * i + 1; // Filho esquerdo
    int right = 2 * i + 2; // Filho direito

    metrics.comparisons++; // Comparação entre filho esquerdo e raiz
    // Se o filho esquerdo é maior que a raiz
    if (left < n && arr[left] > arr[largest])
        largest = left;

    metrics.comparisons++; // Comparação entre filho direito e o maior até agora
    // Se o filho direito é maior que o maior até agora
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Se o maior não é a raiz
    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        metrics.movements++; // Incrementa a contagem de movimentos
        // Recursivamente ajusta o heap afetado
        heapify(arr, n, largest, metrics);
    }
}

// O Heap Sort primeiro constrói um heap max e depois troca o elemento raiz (o maior)
// com o último elemento do heap. Este processo é repetido para o restante do heap.
Metrics heapSort(std::vector<int>& arr) {
    Metrics metrics = {0, 0, 0.0};
    int n = arr.size();
    auto start = std::chrono::high_resolution_clock::now(); // Inicia a contagem do tempo

    // Constrói o heap max
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i, metrics);

    // Extrai os elementos do heap um por um
    for (int i = n - 1; i >= 0; i--) {
        // Move a raiz atual para o fim
        std::swap(arr[0], arr[i]);
        metrics.movements++; // Incrementa a contagem de movimentos
        // Chama heapify no heap reduzido
        heapify(arr, i, 0, metrics);
    }

    auto end = std::chrono::high_resolution_clock::now(); // Finaliza a contagem do tempo
    metrics.time_us =
        std::chrono::duration_cast<std::chrono::microseconds>(end - start).count(); // Calcula o tempo de execução
    return metrics;
}
```

A.4 Merge Sort

A.4.1 MergeSort.h

```
#ifndef MERGESORT_H
#define MERGESORT_H

#include <vector>
#include "Metrics.h"

// Função para realizar a ordenação usando o Merge Sort
// Complexidade de tempo no pior caso: O(n log n)
// Complexidade de tempo no melhor caso: O(n log n)
// Melhor para: listas grandes onde a estabilidade é importante
Metrics mergeSort(std::vector<int>& arr);

#endif // MERGESORT_H
```

A.4.2 MergeSort.cpp

```
#include "MergeSort.h"
#include <chrono>

// Função auxiliar para mesclar duas metades
void merge(std::vector<int>& arr, int left, int mid, int right, Metrics&
    metrics) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Cria arrays temporários
    std::vector<int> L(n1), R(n2);

    // Copia os dados para os arrays temporários
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; ++i)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    // Mescla os arrays temporários de volta ao array original
    while (i < n1 && j < n2) {
        metrics.comparisons++; // Incrementa a contagem de comparações
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            ++i;
        } else {
            arr[k] = R[j];
            ++j;
        }
        metrics.movements++; // Incrementa a contagem de movimentos
        ++k;
    }

    // Copia os elementos restantes de L[], se houver
    while (i < n1) {
        arr[k] = L[i];
        metrics.movements++; // Incrementa a contagem de movimentos
        ++i;
        ++k;
    }

    // Copia os elementos restantes de R[], se houver
    while (j < n2) {
        arr[k] = R[j];
```

```
44          metrics.movements++; // Incrementa a contagem de movimentos
45          ++j;
46          ++k;
47      }
48  }
49
50  // O Merge Sort divide o array em duas metades, ordena cada metade
        recursivamente
51  // e depois mescla as duas metades ordenadas
52  void mergeSortHelper(std::vector<int>& arr, int left, int right, Metrics&
        metrics) {
53      if (left >= right) {
54          return;
55      }
56      int mid = left + (right - left) / 2;
57      mergeSortHelper(arr, left, mid, metrics);
58      mergeSortHelper(arr, mid + 1, right, metrics);
59      merge(arr, left, mid, right, metrics);
60  }
61
62  Metrics mergeSort(std::vector<int>& arr) {
63      Metrics metrics = {0, 0, 0.0};
64      auto start = std::chrono::high_resolution_clock::now(); // Inicia a
            contagem do tempo
65
66      mergeSortHelper(arr, 0, arr.size() - 1, metrics);
67
68      auto end = std::chrono::high_resolution_clock::now(); // Finaliza a
            contagem do tempo
69      metrics.time_us =
            std::chrono::duration_cast<std::chrono::microseconds>(end -
            start).count(); // Calcula o tempo de execução
70      return metrics;
71  }
```

A.5 Quick Sort

A.5.1 QuickSort.h

```
1  #ifndef QUICKSORT_H
2  #define QUICKSORT_H
3
4  #include <vector>
5  #include "Metrics.h"
6
7  // Função para realizar a ordenação usando o Quick Sort
8  // Complexidade de tempo no pior caso: O(n^2)
9  // Complexidade de tempo no melhor caso: O(n log n)
10 // Melhor para: listas grandes onde a estabilidade não é uma preocupação
11 Metrics quickSort(std::vector<int>& arr);
12
13 #endif // QUICKSORT_H
```

A.5.2 QuickSort.cpp

```
1  #include "QuickSort.h"
2  #include <chrono>
3  #include <stack>
4
5  // Função auxiliar para particionar o array
6  int partition(std::vector<int>& arr, int low, int high, Metrics& metrics) {
7      int pivot = arr[high]; // Pivô
8      int i = low - 1; //  ndice  do menor elemento
9      for (int j = low; j < high; ++j) {
```

```
10          metrics.comparisons++; // Incrementa a contagem de comparações
11          // Se o elemento atual é menor ou igual ao pivô
12          if (arr[j] < pivot) {
13              ++i;
14              std::swap(arr[i], arr[j]);
15              metrics.movements++; // Incrementa a contagem de movimentos
16          }
17      }
18      std::swap(arr[i + 1], arr[high]);
19      metrics.movements++; // Incrementa a contagem de movimentos
20      return i + 1;
21  }
22
23  // Versão iterativa do Quick Sort
24  void quickSortIterative(std::vector<int>& arr, int low, int high, Metrics&
        metrics) {
25      // Cria uma pilha auxiliar para armazenar os índices de sub-arrays
26      std::stack<std::pair<int, int>> stack;
27      stack.push(std::make_pair(low, high));
28
29      while (!stack.empty()) {
30          low = stack.top().first;
31          high = stack.top().second;
32          stack.pop();
33
34          // Particiona o array
35          int pi = partition(arr, low, high, metrics);
36
37          // Se houver elementos à esquerda do pivô, adiciona ao stack
38          if (pi - 1 > low) {
39              stack.push(std::make_pair(low, pi - 1));
40          }
41
42          // Se houver elementos à direita do pivô, adiciona ao stack
43          if (pi + 1 < high) {
44              stack.push(std::make_pair(pi + 1, high));
45          }
46      }
47  }
48
49  Metrics quickSort(std::vector<int>& arr) {
50      Metrics metrics = {0, 0, 0.0};
51      auto start = std::chrono::high_resolution_clock::now(); // Inicia a
            contagem do tempo
52
53      quickSortIterative(arr, 0, arr.size() - 1, metrics);
54
55      auto end = std::chrono::high_resolution_clock::now(); // Finaliza a
            contagem do tempo
56      metrics.time_us =
            std::chrono::duration_cast<std::chrono::microseconds>(end -
            start).count(); // Calcula o tempo de execução
57      return metrics;
58  }
```

A.6 Shell Sort

A.6.1 ShellSort.h

```
1  #ifndef SHELLSORT_H
2  #define SHELLSORT_H
3
4  #include <vector>
5  #include "Metrics.h"
```

```
6
7  // Função para realizar a ordenação usando o Shell Sort
8  // Complexidade de tempo no pior caso: Depende da sequência de gaps escolhida,
       mas tipicamente entre O(n^2) e O(n^(3/2))
9  // Complexidade de tempo no melhor caso: O(n log n)
10 // Melhor para: listas de tamanho médio a grande com uma boa escolha de gaps
11 Metrics shellSort(std::vector<int>& arr);
12
13 #endif // SHELLSORT_H
```

A.6.2 ShellSort.cpp

```
1  #include "ShellSort.h"
2  #include <chrono>
3
4  // O Shell Sort é uma generalização do Insertion Sort que permite a troca de
       elementos distantes.
5  // A ideia principal é inicializar o intervalo entre os elementos a serem
       comparados com um valor grande (gap)
6  // e depois reduzir gradualmente esse valor até 1. Isso ajuda a reduzir
       drasticamente o número total de movimentações necessárias.
7  Metrics shellSort(std::vector<int>& arr) {
8      Metrics metrics = {0, 0, 0.0};
9      int n = arr.size();
10     auto start = std::chrono::high_resolution_clock::now(); // Inicia a
           contagem do tempo
11
12     // Inicializa o gap. Aqui estamos usando a sequência de gaps de Shell,
           onde o gap inicial é n/2
13     for (int gap = n / 2; gap > 0; gap /= 2) {
14         // Para cada gap, fazemos uma ordenação por inserção nos elementos que
               estão a essa distância
15         for (int i = gap; i < n; ++i) {
16             int temp = arr[i];
17             int j;
18             // Move os elementos de arr[0..i-gap] que são maiores que temp
19             // para a posição à frente de sua posição atual
20             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
21                 arr[j] = arr[j - gap];
22                 metrics.movements++; // Incrementa a contagem de movimentos
23                 metrics.comparisons++; // Incrementa a contagem de comparações
24             }
25             arr[j] = temp;
26             metrics.movements++; // Incrementa a contagem de movimentos
27             if (j >= gap) {
28                 metrics.comparisons++; // Incrementa a contagem de comparações
29             }
30         }
31     }
32
33     auto end = std::chrono::high_resolution_clock::now(); // Finaliza a
           contagem do tempo
34     metrics.time_us =
           std::chrono::duration_cast<std::chrono::microseconds>(end -
           start).count(); // Calcula o tempo de execução
35     return metrics;
36 }
```

A.7 Main

A.7.1 Main.cpp

```
1  #include <iostream>
2  #include <vector>
```

```cpp
#include <fstream>
#include <chrono>
#include <random>
#include <algorithm>
#include <numeric>
#include "InsertionSort.h"
#include "BubbleSort.h"
#include "HeapSort.h"
#include "MergeSort.h"
#include "QuickSort.h"
#include "ShellSort.h"
#include "Metrics.h"

using namespace std;
using namespace chrono;

// Funções para gerar arranjos
vector<int> generateSortedArray(int size) {
    vector<int> arr(size);
    iota(arr.begin(), arr.end(), 0);
    return arr;
}

vector<int> generateReversedArray(int size) {
    vector<int> arr(size);
    iota(arr.rbegin(), arr.rend(), 0);
    return arr;
}

vector<int> generateAlmostSortedArray(int size) {
    vector<int> arr = generateSortedArray(size);
    random_device rd;
    mt19937 gen(rd());
    for (int i = 0; i < size / 10; ++i) {
        swap(arr[gen() % size], arr[gen() % size]);
    }
    return arr;
}

vector<int> generateRandomArray(int size) {
    vector<int> arr(size);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, size);
    for (int i = 0; i < size; ++i) {
        arr[i] = dis(gen);
    }
    return arr;
}

// Função para testar um algoritmo de ordenação
void testSortingAlgorithm(Metrics (*sortFunction)(vector<int>&), vector<int>&
    arr, ofstream& outputFile, const string& algorithmName, const string&
    arrayType, int size) {
    try {
        cerr << "Testando " << algorithmName << " no array " << arrayType << "
            de tamanho " << size << endl;
        Metrics metrics = sortFunction(arr);
        outputFile << algorithmName << "," << arrayType << "," << size << ","
            << metrics.comparisons << "," << metrics.movements << "," <<
            metrics.time_us << endl;
        cerr << algorithmName << " no array " << arrayType << " de tamanho "
            << size << " completado" << endl;
```

```
60      } catch (const exception& e) {
61          cerr << "Erro ao testar " << algorithmName << " no array " <<
                arrayType << " de tamanho " << size << ": " << e.what() << endl;
62      }
63  }
64
65  int main() {
66      ofstream outputFile("sorting_results.csv");
67      outputFile << "Algorithm,ArrayType,Size,Comparisons,Movements,Time(us)" <<
            endl;
68
69      vector<int> sizes = {10, 100, 1000, 10000, 100000, 1000000};
70      for (int size : sizes) {
71          cerr << "Processando arrays de tamanho " << size << endl;
72
73          // Gerar arrays
74          vector<int> sortedArray = generateSortedArray(size);
75          vector<int> reversedArray = generateReversedArray(size);
76          vector<int> almostSortedArray = generateAlmostSortedArray(size);
77          vector<int> randomArray = generateRandomArray(size);
78
79          // Testar InsertionSort
80          testSortingAlgorithm(insertionSort, sortedArray, outputFile,
                "InsertionSort", "Sorted", size);
81          testSortingAlgorithm(insertionSort, reversedArray, outputFile,
                "InsertionSort", "Reversed", size);
82          testSortingAlgorithm(insertionSort, almostSortedArray, outputFile,
                "InsertionSort", "AlmostSorted", size);
83          testSortingAlgorithm(insertionSort, randomArray, outputFile,
                "InsertionSort", "Random", size);
84
85          // Testar BubbleSort
86          testSortingAlgorithm(bubbleSort, sortedArray, outputFile,
                "BubbleSort", "Sorted", size);
87          testSortingAlgorithm(bubbleSort, reversedArray, outputFile,
                "BubbleSort", "Reversed", size);
88          testSortingAlgorithm(bubbleSort, almostSortedArray, outputFile,
                "BubbleSort", "AlmostSorted", size);
89          testSortingAlgorithm(bubbleSort, randomArray, outputFile,
                "BubbleSort", "Random", size);
90
91          // Testar HeapSort
92          testSortingAlgorithm(heapSort, sortedArray, outputFile, "HeapSort",
                "Sorted", size);
93          testSortingAlgorithm(heapSort, reversedArray, outputFile, "HeapSort",
                "Reversed", size);
94          testSortingAlgorithm(heapSort, almostSortedArray, outputFile,
                "HeapSort", "AlmostSorted", size);
95          testSortingAlgorithm(heapSort, randomArray, outputFile, "HeapSort",
                "Random", size);
96
97          // Testar MergeSort
98          testSortingAlgorithm(mergeSort, sortedArray, outputFile, "MergeSort",
                "Sorted", size);
99          testSortingAlgorithm(mergeSort, reversedArray, outputFile,
                "MergeSort", "Reversed", size);
100         testSortingAlgorithm(mergeSort, almostSortedArray, outputFile,
                "MergeSort", "AlmostSorted", size);
101         testSortingAlgorithm(mergeSort, randomArray, outputFile, "MergeSort",
                "Random", size);
102
103         // Testar QuickSort
104         testSortingAlgorithm(quickSort, sortedArray, outputFile, "QuickSort",
```

```
            "Sorted", size);
105         testSortingAlgorithm(quickSort, reversedArray, outputFile,
                "QuickSort", "Reversed", size);
106         testSortingAlgorithm(quickSort, almostSortedArray, outputFile,
                "QuickSort", "AlmostSorted", size);
107         testSortingAlgorithm(quickSort, randomArray, outputFile, "QuickSort",
                "Random", size);
108
109         // Testar ShellSort
110         testSortingAlgorithm(shellSort, sortedArray, outputFile, "ShellSort",
                "Sorted", size);
111         testSortingAlgorithm(shellSort, reversedArray, outputFile,
                "ShellSort", "Reversed", size);
112         testSortingAlgorithm(shellSort, almostSortedArray, outputFile,
                "ShellSort", "AlmostSorted", size);
113         testSortingAlgorithm(shellSort, randomArray, outputFile, "ShellSort",
                "Random", size);
114
115         // Liberação de memória
116         sortedArray.clear();
117         reversedArray.clear();
118         almostSortedArray.clear();
119         randomArray.clear();
120         sortedArray.shrink_to_fit();
121         reversedArray.shrink_to_fit();
122         almostSortedArray.shrink_to_fit();
123         randomArray.shrink_to_fit();
124
125         cerr << "Processamento de arrays de tamanho " << size << " finalizado"
                << endl;
126     }
127
128     outputFile.close();
129     cerr << "Programa finalizado" << endl;
130     return 0;
131 }
```