Análise de Métodos de Ordenação

Otávio Garcia de Oliveira Farias

Instituto de Ensino Superior – Universidade Federal do Pampa (UNIPAMPA) - Engenharia de Computação

otaviogarcia.aluno@unipampa.edu.br

Resumo: *Este relatório analisa o desempenho de diferentes algoritmos de ordenação, medindo o tempo de execução para várias entradas e tipos de arrays (crescente, decrescente, desordenado e com elementos repetidos). A implementação dos algoritmos foi feita em Java. Foram testados os métodos Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort e Bucket Sort. Para a análise dos resultados, foram utilizadas as ferramentas LibreOffice Calc, para organizar e visualizar o desempenho em planilhas, e Python, para gerar gráficos comparativos entre os métodos. Os resultados ilustram o comportamento dos algoritmos em diferentes cenários, possibilitando uma melhor compreensão das diversas técnicas empregadas na construção e aplicação de algoritmos de ordenação.*

Abstract: *This report analyzes the performance of different sorting algorithms, measuring execution time for various input types and array configurations (increasing order, decreasing order, unsorted, and with repeated elements). The algorithms were implemented in Java, and the methods tested included Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Bucket Sort. For result analysis, LibreOffice Calc was used to organize and visualize performance in spreadsheets, and Python was used to generate comparative graphs between methods. The results illustrate the behavior of the algorithms in different scenarios, enabling a better understanding of the various techniques employed in the construction and application of sorting algorithms.*

Introdução:

A ordenação de dados de diferentes tipos é fundamental em várias aplicações, sendo utilizadas técnicas distintas para alcançar um método eficiente em diversas condições. O tempo de execução dos algoritmos é crucial para múltiplas aplicações; contudo, é necessário entender a relação entre o tempo de execução e o espaço requerido para o algoritmo. Dependendo da quantidade e do tipo de dados, pode-se optar pelo método mais adequado a ser empregado.

Este relatório tem como objetivo analisar o desempenho de métodos de ordenação em cenários distintos, como arrays ordenados em ordem crescente e decrescente,

desordenados e com elementos repetidos. A análise contempla a avaliação para 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de entradas. Para isso, foram estudados os algoritmos Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort e Bucket Sort, todos implementados em Java, utilizando o método Comparable.

Além disso, foi realizada uma comparação entre o algoritmo Merge Sort utilizando o método Comparable, o Merge Sort com o auxílio do Insertion Sort e o Merge Sort com o tipo primitivo int. Também foi necessário comparar o algoritmo QuickSort utilizando recursão e o algoritmo QuickSort simulado com uma estrutura de pilha, sem recursão.

Metodologia:

O objetivo desta metodologia foi desenvolver uma forma eficaz de avaliar os métodos de ordenação, considerando fatores como a quantidade de elementos e o tipo do array de entrada. Para garantir maior consistência nos resultados e mitigar a interferência de variáveis aleatórias, os algoritmos foram testados com as mesmas entradas e executados 30 vezes.

Foram utilizadas classes auxiliares implementadas em Java para facilitar a avaliação dos algoritmos, contendo métodos como *gerarArrayInteger* e *gerarArrayInt*, responsáveis por criar entradas em ordem crescente, decrescente, desordenada e com elementos repetidos. Nos algoritmos *CountingSort* e *RadixSort*, os arrays eram formados por elementos do tipo *int*; para o *BucketSort*, utilizou-se um método auxiliar para gerar um array *float*, equivalente aos utilizados nos testes de algoritmos lineares. Os demais algoritmos foram testados com arrays de *Integer*.

Todos os algoritmos foram implementados em uma classe chamada *Sorting*, e posteriormente chamados por uma classe *TimeTest*, responsável por executar cada função diversas vezes e salvar os resultados em um arquivo *.csv*. Durante a execução da classe *TimeTest*, argumentos eram passados para definir a configuração dos testes: o argumento 0 definia o número de entradas; o argumento 1, a quantidade de repetições; o argumento 2, o tipo de array (0 para crescente, 1 para decrescente, 2 para elementos iguais, 3 para desordenados); no caso do *MergeSort* com apoio do *InsertionSort*, o argumento 4 definia o tamanho do array a partir do qual o *InsertionSort* seria chamado.

A medição do tempo de execução de cada algoritmo era registrada em uma *List* e posteriormente salva em um arquivo, com o nome do método de ordenação

correspondente. Em cada teste, era gerada uma nova *List* para armazenar os resultados. Para medir o tempo, utilizou-se a função *System.nanoTime()* do Java, chamada separadamente para cada algoritmo. Um array auxiliar foi definido para cada método e mantido igual para todos, sendo ordenado por cada algoritmo. Exemplo de código abaixo;

```
Integer[] array = arrayTest.clone();

t_ini = System.nanoTime();

Sorting.(método escolhido)(array, Integer.parseInt(args[0])); t_fim = System.nanoTime();

tempos.add(t_fim - t_ini);
```

Durante os testes, foi necessário substituir o QuickSort recursivo por uma versão não-recursiva que simula uma pilha, devido ao erro de *StackOverflowError* causado pelas diversas chamadas recursivas no QuickSort para entradas com 100.000 elementos. Todos os testes foram então repetidos com o QuickSort sem recursão, e foi realizada uma comparação entre ambas as versões para entradas inferiores a 100.000 elementos.

Foram avaliadas variações do MergeSort, incluindo o MergeSort com apoio do InsertionSort para arrays de 5, 10 e 50 elementos, visando maior eficiência em arrays pequenos. Também foi utilizada uma versão do MergeSort com tipos primitivos *int* em vez de *Comparable*, para avaliar o impacto do uso de *Comparable*, que adiciona versatilidade, mas pode comprometer o desempenho em certos cenários.

Após a geração dos arquivos *.csv* com os tempos de execução dos algoritmos, utilizou-se o LibreOffice Calc para organizar as planilhas e criar gráficos. Para minimizar o impacto de variáveis externas nos resultados, os valores foram tratados com suas médias, excluindo-se os valores mínimo e máximo, o que conferiu mais confiabilidade aos dados.

A geração de gráficos foi realizada em Python com a biblioteca *matplotlib.pyplot*. Foram criados gráficos de dispersão para cada algoritmo individualmente, gráficos de linha para as três implementações do MergeSort, as duas do QuickSort e para todos os algoritmos aplicados a arrays desordenados com entradas de 10, 100, 1.000, 10.000, 100.000 e 1.000.000. Além disso, foram gerados gráficos de barras para entradas de 10.000 elementos, comparando o desempenho dos algoritmos em arrays ordenados em ordem crescente, decrescente e com elementos iguais. Cada tamanho de entrada para arrays desordenados também resultou em um gráfico de barras correspondente. Com isso, foi possível visualizar o comportamento dos algoritmos em diferentes condições.

Métodos de ordenação:

InsertionSort: Esse algoritmo funciona com base na ideia de construir uma lista auxiliar e inserir os elementos da lista original já nas suas respectivas posições ideais. Esse algoritmo tem complexidade O(n2)O(n^2)O(n2), sendo eficiente para pequenas entradas. Na sua implementação, dois loops são responsáveis por comparar os elementos e simular a inserção em uma nova lista: o primeiro loop seleciona um valor, e o segundo o compara com os demais elementos, colocando-o na posição ideal.

BubbleSort: O algoritmo compara todos os valores com suas posições adjacentes, modificando-as caso necessário. Como esse processo é aplicado a todos os elementos, os maiores valores “flutuam” até o fim da lista, ficando na sua posição ideal. Possui complexidade O(n2)O(n^2)O(n2); apesar de simples na implementação, é pouco eficiente para listas grandes.

SelectionSort: O algoritmo percorre a lista de valores em busca do menor elemento e, ao encontrá-lo, o coloca na primeira posição. O mesmo ocorre para o segundo elemento, e o processo se repete até todos os elementos estarem ordenados. Possui complexidade O(n2)O(n^2)O(n2), constante para qualquer caso, o que o torna pouco eficiente para grandes quantidades de dados.

ShellSort: Uma versão otimizada do Insertion Sort, que inicialmente compara elementos entre si com uma certa distância, diminuindo gradativamente essa distância. Dessa forma, quando a comparação ocorre entre elementos adjacentes, a lista já está mais ordenada, tornando o processo mais eficiente. Tem complexidade O(nlogn)O(n \log n)O(nlogn) e é eficaz para listas de tamanho médio; a escolha da distância inicial de comparação pode impactar a eficiência do algoritmo.

HeapSort: Utiliza uma estrutura de árvore binária completa para a ordenação. Inicialmente, é construída uma árvore de máximos (max-heap). Após a criação da árvore através de chamadas recursivas da função responsável por sua construção, os elementos são retirados da árvore e colocados nas suas posições ordenadas. Tem complexidade O(nlogn)O(n \log n)O(nlogn) para todos os casos, mas não é estável devido às modificações realizadas na árvore.

MergeSort: É um algoritmo de divisão e conquista que separa a lista em duas partes e repete o processo até que as partes sejam pequenas o suficiente para serem ordenadas. Em seguida, as listas já ordenadas são mescladas, formando uma lista totalmente ordenada. Pode-se realizar a ordenação com o próprio algoritmo Merge para sublistas de tamanho dois ou usar métodos auxiliares, como o Insertion Sort, para listas pequenas, aumentando assim a eficiência.

QuickSort (com recursão): Segue o princípio de divisão e conquista, escolhendo um pivô e posicionando-o na sua posição ideal. Após isso, a lista é dividida em elementos maiores e menores que o pivô, e para ambas as partes são feitas

chamadas recursivas do método. Esse processo é repetido até que a lista esteja totalmente ordenada. O algoritmo é eficiente na maioria dos casos, exceto quando o pivô é o valor mínimo ou máximo da lista. Contudo, não é estável, e a quantidade de chamadas recursivas pode sobrecarregar a pilha de recursão.

QuickSort (sem recursão): Segue a mesma ideia do QuickSort com recursão, mas utiliza uma estrutura de pilha para simular as chamadas recursivas. Evita o estouro de pilha, mas possui uma implementação mais complexa e com valores constantes maiores.

CountingSort: Ordena valores inteiros contando a frequência de cada elemento e determinando a sua posição final. É um algoritmo de complexidade linear, sendo extremamente eficiente para grandes quantidades de dados. Porém, dependendo do intervalo dos valores, pode consumir bastante espaço.

RadixSort: Ordena elementos a partir dos seus dígitos, selecionando um dígito por vez e usando o Counting Sort para ordená-los. O processo é repetido para os demais dígitos. É estável e, por ordenar apenas dígitos de 0 a 9, ao chamar o Counting Sort, cria listas menores, consumindo menos espaço, o que o torna extremamente eficiente.

BucketSort: Distribui os elementos em pequenos grupos. Cada grupo é ordenado através do Insertion Sort e, posteriormente, os grupos ("baldes") são concatenados. É eficaz para elementos bem distribuídos em um intervalo e rápido para grandes entradas.

Explorando as implementações:

Análise de Resultados:

Para entradas de 10 elementos, observou-se que algoritmos como CountingSort, BucketSort e QuickSort(sem recursão) apresentaram desempenho inferior ao esperado. Isso ocorreu devido aos altos valores constantes associados a esses métodos, o que pode torná-los menos eficientes para pequenas quantidades de dados, onde o tempo de execução é mais sensível a sobrecargas internas do algoritmo.

Por outro lado, algoritmos mais simples, como BubbleSort e InsertionSort, que possuem complexidade $O(n^2)$, mostraram-se mais eficientes para arrays pequenos. A simplicidade dessas implementações e a baixa sobrecarga de execução compensam a complexidade quadrática, tornando-os adequados para tamanhos de entrada reduzidos.

HeapSort, MergeSort e RadixSort também demonstraram desempenho razoável para 10 elementos, posicionando-se entre os algoritmos de alta sobrecarga e aqueles com execução mais simples. Esses resultados indicam que, para pequenas entradas, algoritmos mais complexos podem ser desvantajosos em comparação com métodos mais simples, pois o custo constante e a sobrecarga de operações internas impactam significativamente o tempo de execução.

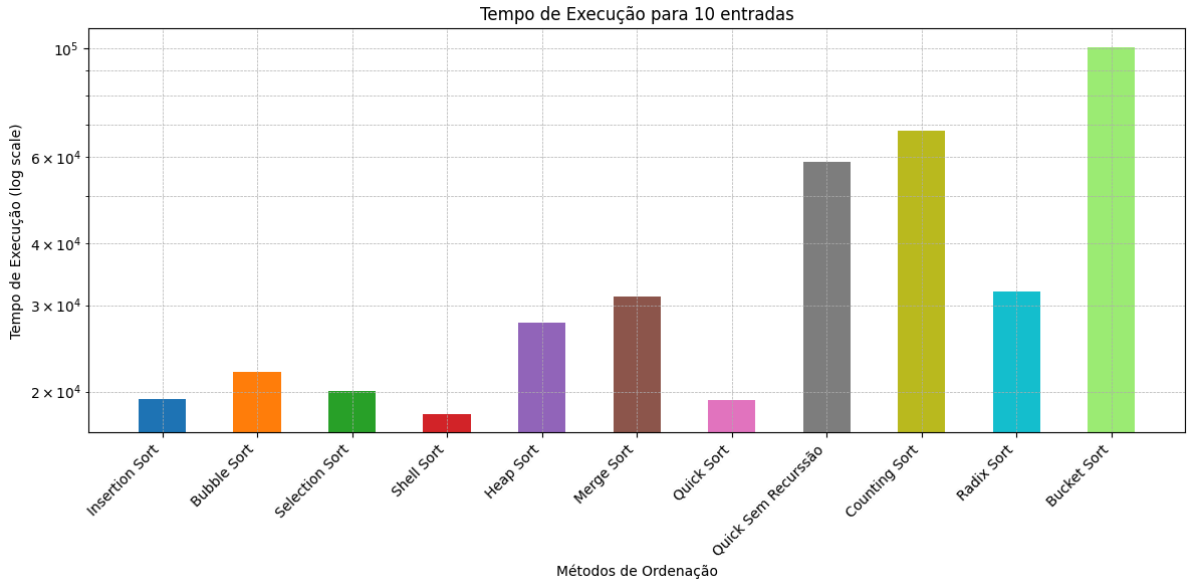

Figura 1: Gráfico de desempenho dos algoritmos de ordenação de 10 entradas. Fonte: Elaborado pelo autor.

A partir de 100 entradas, torna-se visível o impacto da complexidade dos algoritmos no tempo de execução. InsertionSort, BubbleSort e SelectionSort apresentaram os maiores tempos de execução, juntamente com BucketSort e QuickSort (sem recursão), cujos altos fatores constantes tornam esses algoritmos ineficientes para entradas pequenas.

O algoritmo QuickSort se destacou devido à sua complexidade O(n log n) e baixos valores constantes, sendo extremamente eficiente para entradas desordenadas, os demais algoritmos mantiveram desempenho consistente devido à sua complexidade mais baixa em comparação com os algoritmos quadráticos

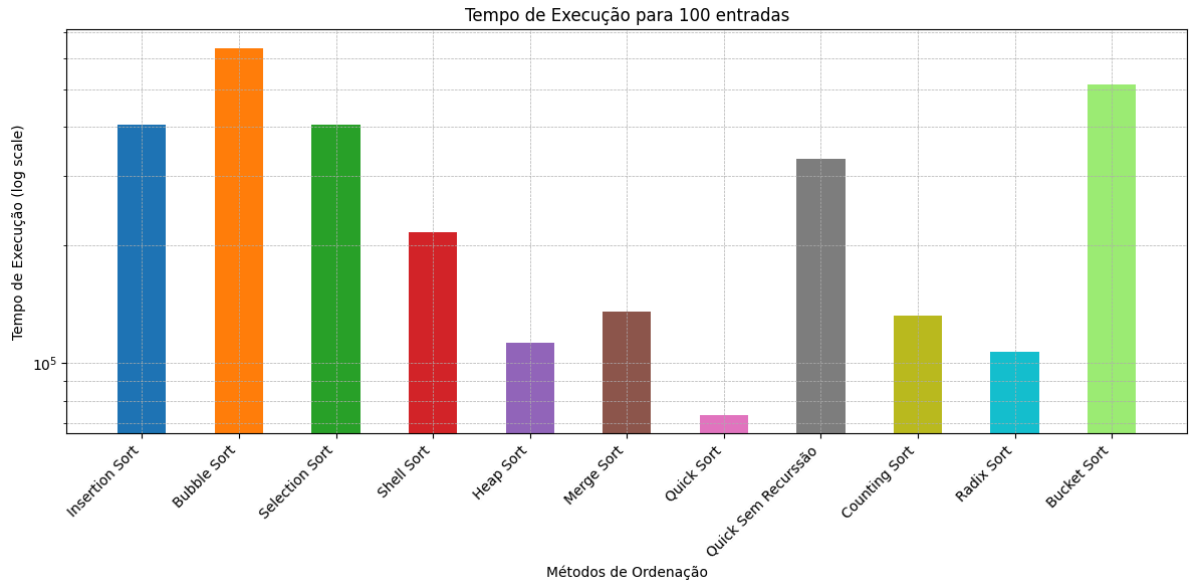

Figura 2: Gráfico de desempenho dos algoritmos de ordenação de 100 entradas. Fonte: Elaborado pelo autor.

O comportamento dos algoritmos com complexidade O(n²) e O(n log n) para 1.000 entradas mantém-se semelhante ao observado com 100 entradas. Destaca-se o equilíbrio das implementações do QuickSort: apesar dos valores constantes mais elevados na versão sem recursão, esse fator tem menor impacto no tempo total para entradas maiores.

Os métodos de ordenação lineares, embora apresentem valores constantes altos, tiveram desempenho significativamente melhor que os demais algoritmos. Isso se deve à sua complexidade O(n), o que os torna mais rápidos que os algoritmos O(n log n) para grandes entradas, mesmo com menor versatilidade. Essa menor versatilidade ocorre devido à especialização desses métodos para tipos de dados, enquanto algoritmos como MergeSort e QuickSort são mais adaptáveis a diferentes contextos de ordenação.

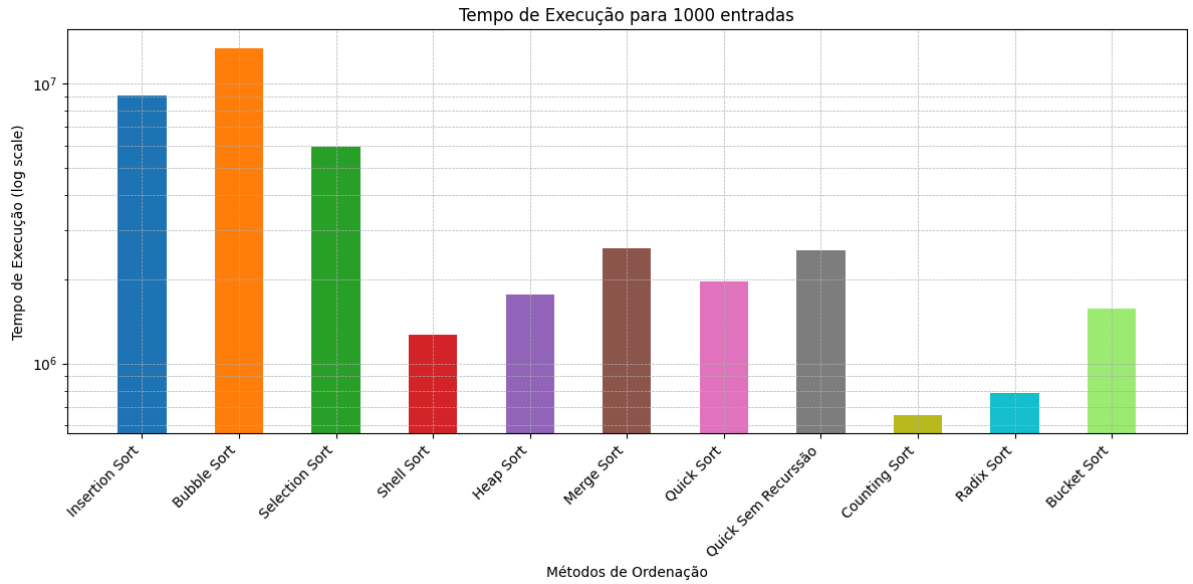

Figura 3: Gráfico de desempenho dos algoritmos de ordenação de 1.000 entradas. Fonte: Elaborado pelo autor.

Com 10.000 entradas, torna-se evidente a eficiência dos algoritmos com complexidade O(n log n) e, especialmente, O(n). Apesar dos fatores constantes mais elevados, esses algoritmos mantêm tempos de execução baixos, o que indica sua adequação para grandes volumes de dados, onde a complexidade inferior compensa a sobrecarga inicial.

Entre os algoritmos analisados, QuickSort (sem recursão) e BucketSort se destacaram pela rapidez na execução, enquanto CountingSort, com sua complexidade O(n) e baixos valores constantes, foi o mais rápido em geral. Os métodos de complexidade O(n log n), como MergeSort e HeapSort, mostraram tempos de execução muito semelhantes entre si, confirmando sua estabilidade para grandes entradas.

Por outro lado, os algoritmos de complexidade $O(n^2)$ — InsertionSort, BubbleSort e SelectionSort — mostraram-se ineficazes para ordenar 10.000 valores, com tempo de execução aumentando exponencialmente, um comportamento que tende a se intensificar para entradas ainda maiores.

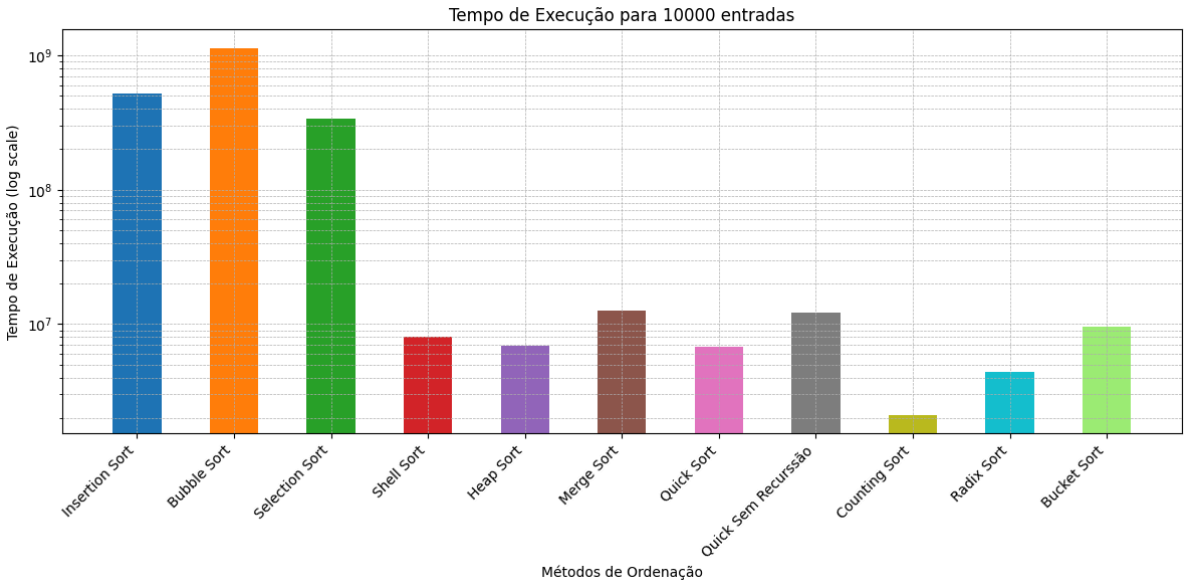

Figura 4: Gráfico de desempenho dos algoritmos de ordenação de 10.000 entradas. Fonte: Elaborado pelo autor.

Durante os testes com 100.000 entradas, o algoritmo QuickSort (com recursão) provocou o erro StackOverflowError, inviabilizando seu uso nos testes subsequentes. Esse erro possivelmente ocorreu devido ao grande número de chamadas recursivas realizadas. Dependendo do valor do pivô escolhido, as chamadas recursivas podem chegar a ser feitas até n-1 vezes, preenchendo assim a pilha de execução. Por conta desse problema, foi necessária a implementação de um QuickSort sem recursão, que simulou uma pilha de execução e se mostrou o mais rápido para 100.000 entradas. Para garantir uma análise completa, posteriormente foi realizada uma comparação entre as versões do QuickSort utilizadas.

As implementações com complexidade $O(n^2)$ mostraram-se novamente ineficientes. Por outro lado, ShellSort, HeapSort e MergeSort foram extremamente eficientes devido à sua complexidade mais baixa e aos menores valores constantes. Os algoritmos BucketSort, RadixSort e CountingSort apresentaram desempenho levemente inferior em relação aos algoritmos de complexidade O(n log n). Como observado em testes com entradas menores, seus fatores constantes tornaram-se menos relevantes para entradas grandes. A leve diferença de desempenho pode estar relacionada ao fato de que os testes não foram executados com os mesmos arrays de entrada, pois esses algoritmos foram implementados para tipos de dados específicos: float para BucketSort, int para RadixSort e CountingSort, ao invés de Comparable, o que resultou em testes diferentes dos demais algoritmos.

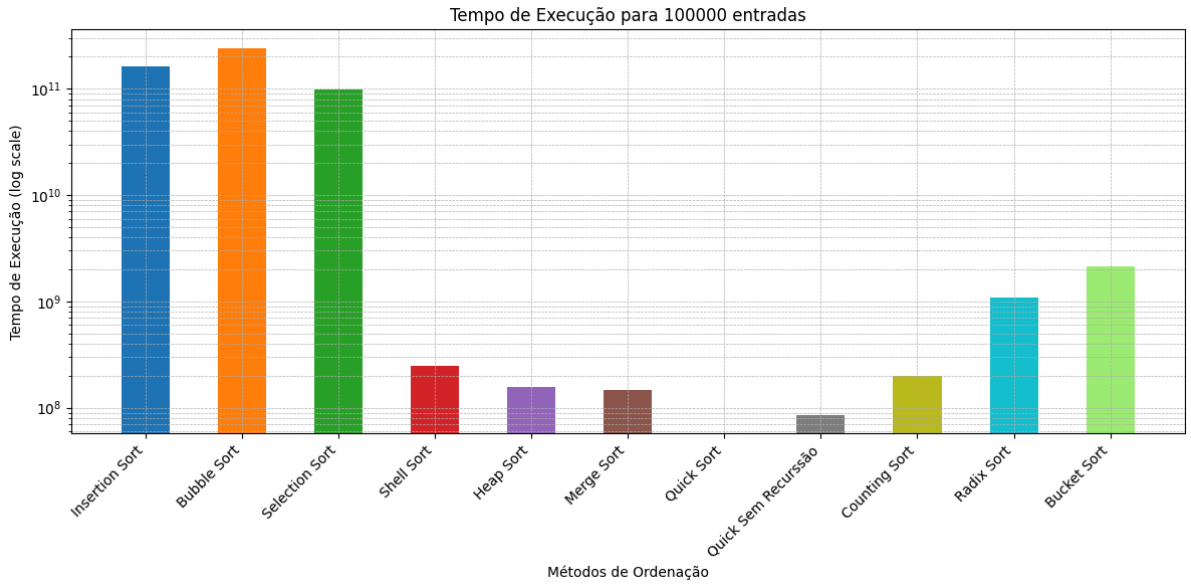

Figura 5: Gráfico de desempenho dos algoritmos de ordenação de 100.000 entradas. Fonte: Elaborado pelo autor.

Com 1.000.000 de entradas, o tempo de execução dos algoritmos aumentou consideravelmente, e o InsertionSort, MergeSort e BubbleSort permaneceram em execução por mais de 12 horas sem completar a ordenação, inviabilizando a coleta dos seus dados. Esse longo tempo de execução pode ser explicado pela complexidade $O(n^2)$, que exigiria cerca de 1 trilhão de operações para ordenar essa quantidade de elementos, desconsiderando os fatores constantes adicionais de cada algoritmo. Assim, fica evidente a ineficácia desses métodos para valores de entrada elevados.

Por outro lado, os algoritmos com complexidade O(n log n) foram extremamente eficazes, sendo possível executar todos os testes. Devido à sua complexidade, esses algoritmos necessitam de aproximadamente 19.931.569 operações para concluir a execução, um número significativamente menor em comparação ao InsertionSort, BubbleSort e SelectionSort.

Os métodos lineares — BucketSort, RadixSort e CountingSort — foram os mais rápidos, com destaque para o CountingSort, que é utilizado na execução do RadixSort, sendo o mais veloz dos três. Portanto, conforme a quantidade de entradas aumentava, os fatores constantes tornaram-se menos relevantes, tornando os algoritmos lineares mais eficazes, ainda que com menor versatilidade.

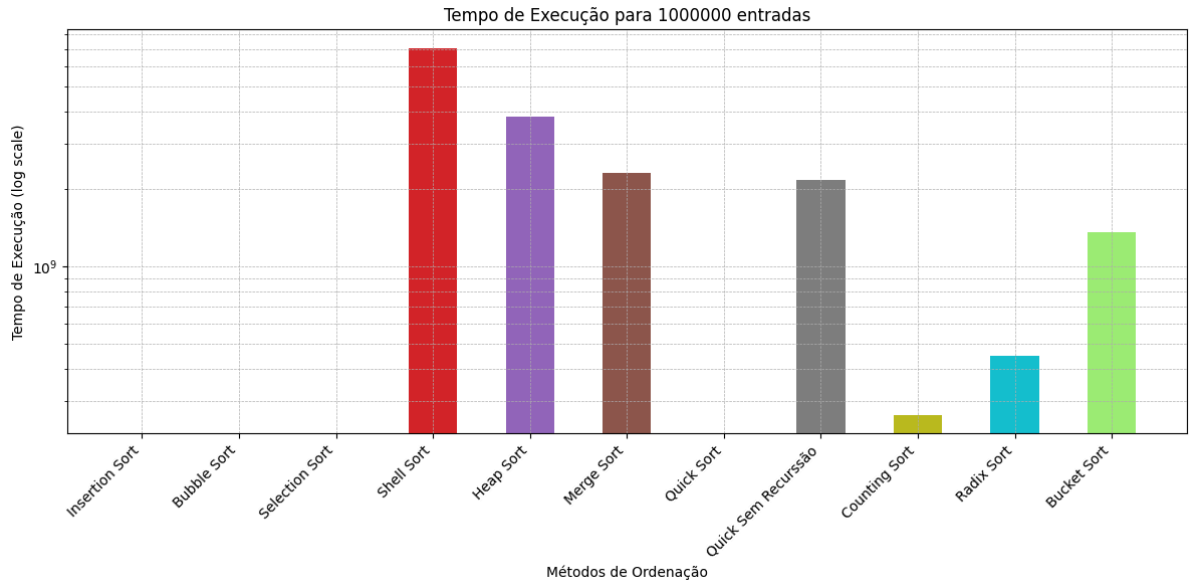

Figura 6: Gráfico de desempenho dos algoritmos de ordenação de 1.000.000 entradas. Fonte: Elaborado pelo autor.

Juntamente com o teste geral com arrays desordenados, foram realizados testes com arrays em ordem crescente, decrescente e com elementos iguais, para avaliar a corretude e o comportamento de cada método nesses cenários.

Para arrays em ordem crescente (já ordenados), o algoritmo InsertionSort se destacou, pois realiza n-1 comparações e não efetua trocas, sendo, portanto, muito eficiente nesse caso. Em contrapartida, nos algoritmos BubbleSort e SelectionSort, ainda são realizadas diversas comparações mesmo com o array já ordenado.

Nos algoritmos QuickSort (tanto na versão com simulação de pilha quanto na recursiva), um array já ordenado impacta diretamente o desempenho, pois, ao escolher o primeiro elemento como pivô, a partição cria um array com n-1 elementos, resultando no pior caso. Por isso, o QuickSort se mostra mais eficiente para arrays desordenados.

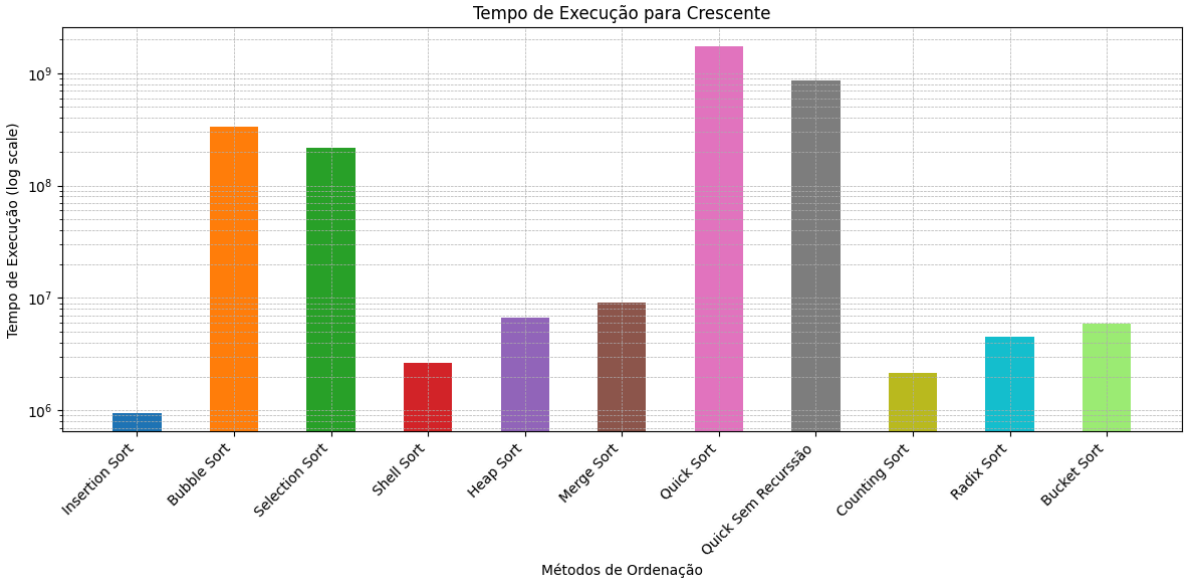

Figura 7: Gráfico de desempenho dos algoritmos de ordenação com entradas em ordem crescente. Fonte: Elaborado pelo autor.

Os algoritmos InsertionSort, SelectionSort, BubbleSort e ambos os métodos QuickSort apresentam o pior caso nos testes para entradas desordenadas, pois particionam o array em n-1 elementos. Os demais algoritmos mantêm seu comportamento consistente em comparação com entradas desordenadas.

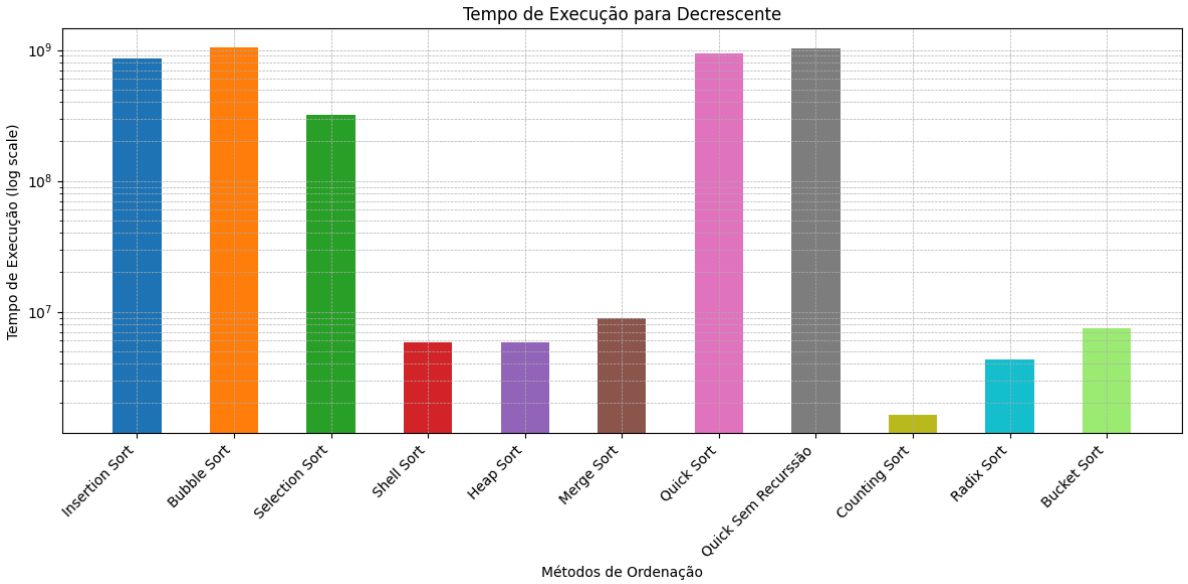

Figura 8: Gráfico de desempenho dos algoritmos de ordenação com entradas em ordem decrescente. Fonte: Elaborado pelo autor.

Já para entradas com o mesmo valor, o desempenho, de forma geral, é similar ao dos arrays em ordem crescente, pois ambos já se encontram ordenados. Pode-se destacar o desempenho do HeapSort, que necessita fazer menos modificações na árvore, ganhando assim eficiência.

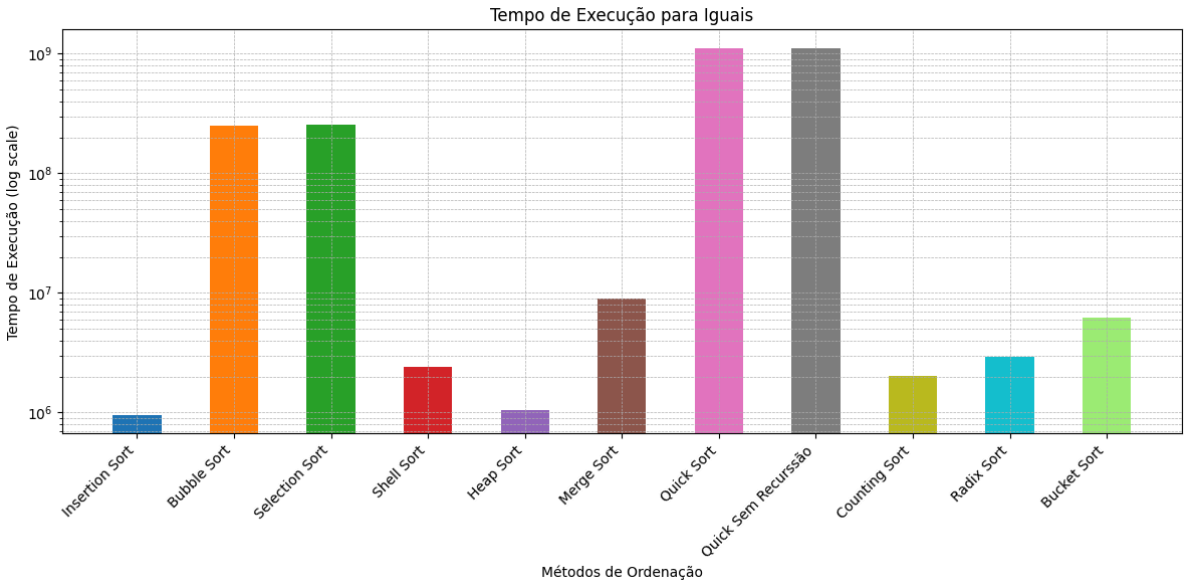

Figura 9: Gráfico de desempenho dos algoritmos de ordenação com entradas repetidas. Fonte: Elaborado pelo autor.

Devido à mudança necessária de um QuickSort com recursão para um sem recursão, é interessante comparar o comportamento de ambos para as mesmas quantidades de entradas. Por causa de fatores como a necessidade de criar uma estrutura de pilha e lidar com a inserção e retirada de elementos, o QuickSort com recursão acaba apresentando valores constantes maiores do que a implementação sem recursão. Assim, em valores absolutos, o QuickSort com recursão é 40% mais rápido, porém pode causar o enchimento da pilha de execução, resultando em erros. Conforme o número de entradas aumenta, a tendência é que os fatores constantes influenciem menos, aproximando os tempos de execução.

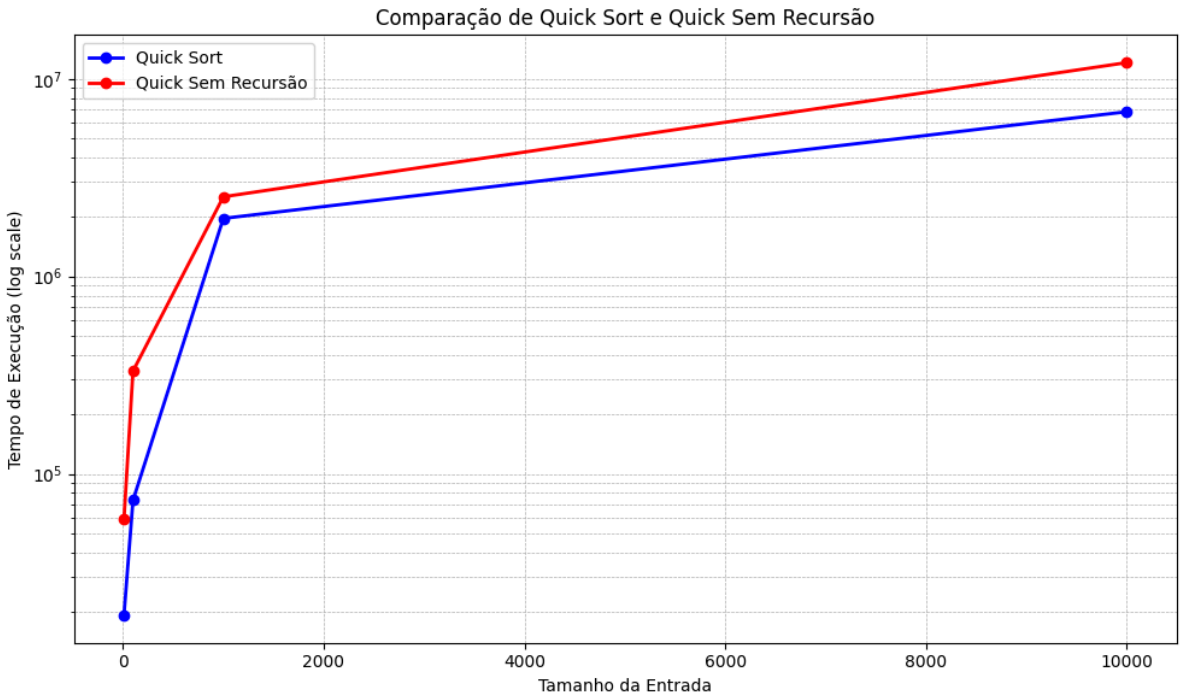

Figura 10: Gráfico de desempenho dos algoritmos QuickSort. Fonte: Elaborado pelo autor.

Foram testadas diferentes versões do *MergeSort* para observar seu comportamento em três implementações: com o uso de *Comparable* (assim como os demais algoritmos), com o auxílio do *InsertionSort* para arrays pequenos (tamanhos de 5, 10 e 50 elementos) e com o tipo *int*.

De modo geral, a implementação com *InsertionSort* acionado para arrays de 5 elementos mostrou-se mais eficiente que as demais, sendo 9% mais rápida que a implementação original. Em seguida, os tamanhos de 10 e 50 elementos trouxeram aumentos de eficiência de 7,97% e 8,69%, respectivamente. Esse método é efetivo pois, para poucas entradas, o *MergeSort* é naturalmente mais lento que o *InsertionSort*.

A implementação com o tipo *int*, ao invés de *Comparable*, mostrou uma melhoria impressionante de 78% no tempo de execução, evidenciando o custo em desempenho ao tornar os métodos de ordenação mais versáteis. Assim, ao priorizar a eficiência, é preferível escolher um método específico em detrimento de um mais genérico.

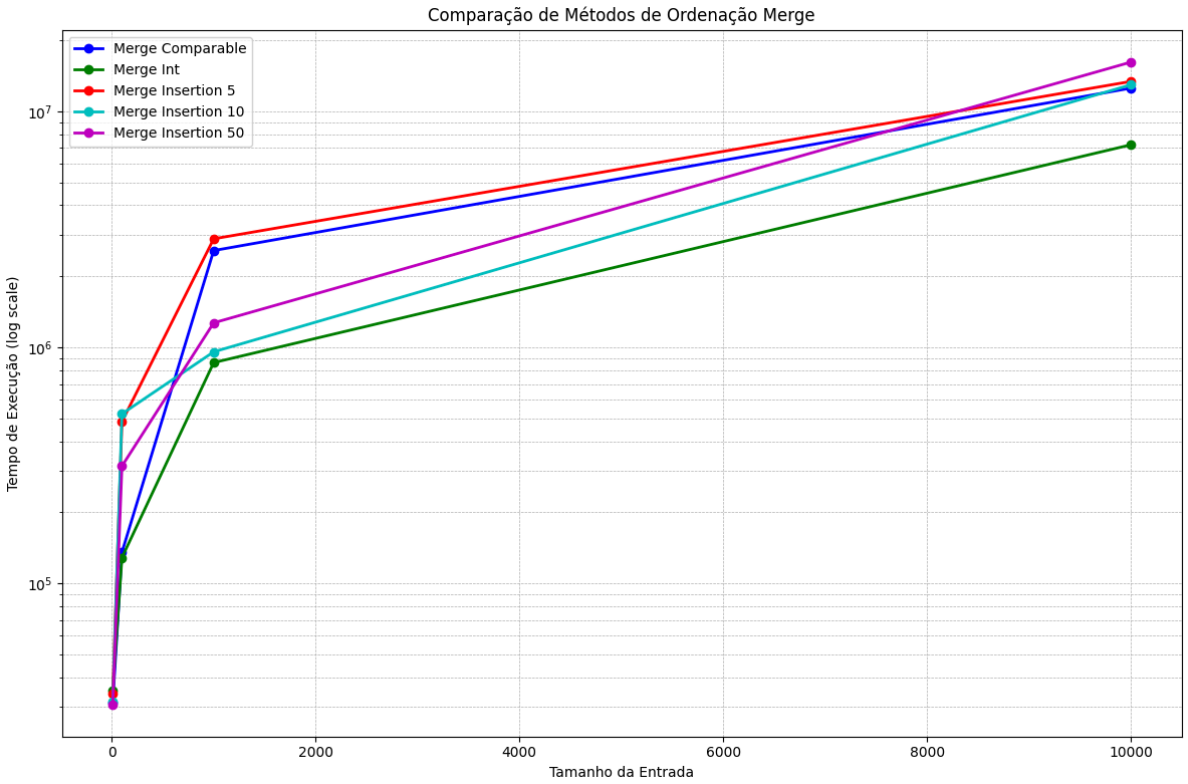

Figura 11: Gráfico de desempenho dos algoritmos MergeSort. Fonte: Elaborado pelo autor.

Em uma comparação geral dos algoritmos, é possível observar um padrão de comportamento: algoritmos com complexidade da ordem de $O(n^2)$, por serem mais simples, mostram-se mais eficazes para pequenas entradas. Em contrapartida, implementações de complexidade O(n) tornam-se mais eficientes para entradas maiores. No entanto, é importante considerar o uso de memória adicional gerado pela criação de arrays auxiliares no *CountingSort*, o que pode influenciar a escolha desses métodos, além de serem menos genéricos. Já os algoritmos de complexidade O(n log n) apresentam um equilíbrio entre fatores como valores constantes, complexidade e versatilidade.

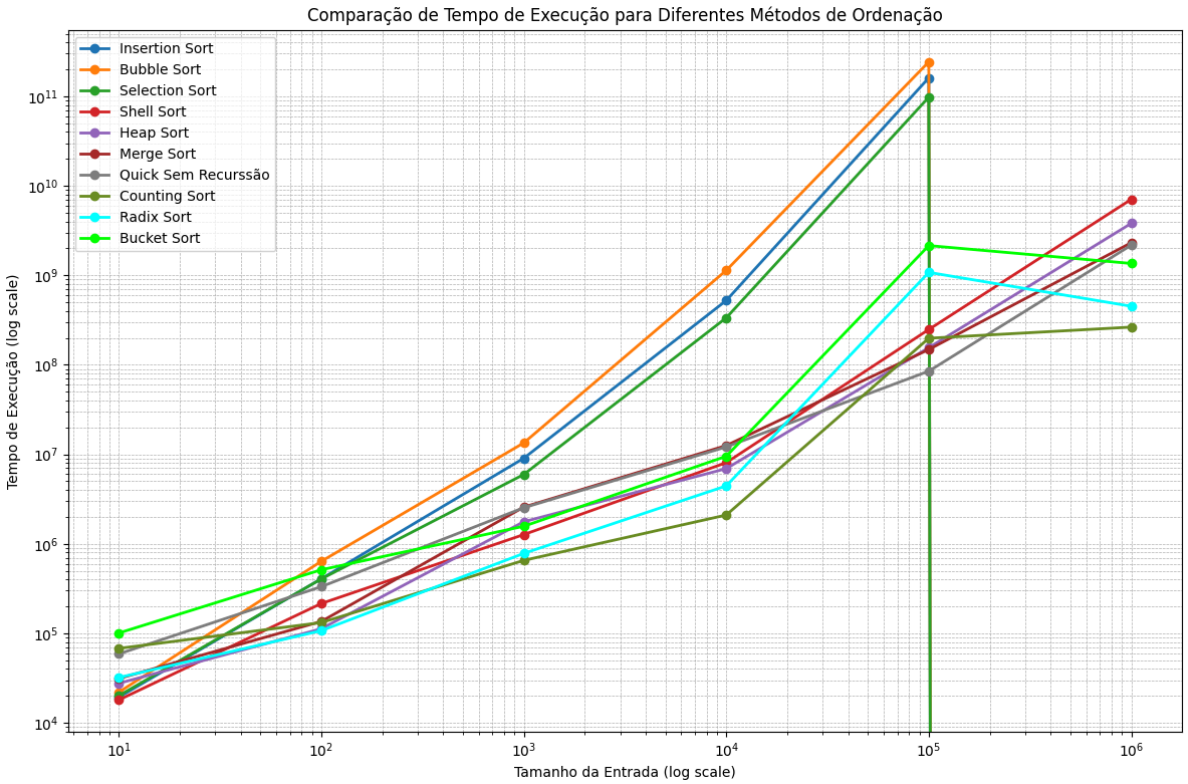

Figura 12: Gráfico de desempenho de todos os algoritmos de ordenação. Fonte: Elaborado pelo autor.

Conclusão:

A partir da análise dos diferentes tipos de algoritmos, pode-se perceber que cada método é indicado para uma determinada tarefa. Além disso, é necessário compreender o desempenho de cada um em relação ao tempo e ao espaço. Para entradas pequenas, o algoritmo InsertionSort se destaca por sua simplicidade e baixo uso de memória. Para entradas de tamanho médio, destaca-se o QuickSort, que, dependendo da escolha do pivô, pode ser extremamente eficiente. No entanto, para listas com muitos elementos já ordenados, ele se torna ineficiente, além de fazer grande uso da pilha de execução devido ao número elevado de recursões.

O algoritmo ShellSort também apresenta bons resultados para entradas de tamanho médio, além de utilizar pouco espaço. Contudo, de forma geral, ele é mais lento que o QuickSort. Quando se trata de entradas grandes, algoritmos de complexidade linear são ótimas escolhas, pois, nesse caso, seus fatores constantes tornam-se menos relevantes. No entanto, esses algoritmos são limitados em relação ao tipo de dado que conseguem ordenar, sendo o CountingSort o mais eficiente, apesar do seu elevado uso de espaço.

Outro fator importante é a estabilidade do método escolhido. Algoritmos como HeapSort, QuickSort e ShellSort não são estáveis, sendo inviável utilizá-los para ordenar listas que necessitam manter uma ordenação prévia. O uso da interface Comparable é extremamente interessante, pois permite generalizar o tipo de dado a ser comparado; contudo, conforme observado nos testes com diferentes variações do MergeSort, essa abordagem afeta negativamente o desempenho do método.

Portanto, por meio da análise dos diferentes métodos de ordenação, fica evidente que diferentes abordagens para realizar a mesma tarefa afetam drasticamente os resultados e a eficácia dos algoritmos. Assim, é preferível realizar uma análise criteriosa sobre quais métodos utilizar e como implementá-los para resolver o problema proposto.

Referências bibliográficas:

https://matplotlib.org/stable/users/index.html
CORMEN, Thomas H.; LEISERSON, Charles E.; Ronald L. Rivest; et al. Algoritmos: teoria e prática. 4. Rio de Janeiro: GEN LTC, 2024. E-book. ISBN 9788595159914.

Anexos:

Código com os algoritmos de ordenação(Sorting.java):

```
package sort;
import java.util.Stack;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class Sorting {

   // Insertion Sort
   public static <T extends Comparable<T>> void insertion(T[] A, int n) {
      for (int i = 1; i < n; i++) {
         T key = A[i];
         int j = i - 1;

         while (j >= 0 && A[j].compareTo(key) > 0) {
            A[j + 1] = A[j];
            j--;
         }
         A[j + 1] = key;
      }
   }

   // Bubble Sort
   public static <T extends Comparable<T>> void bubble(T[] A, int n) {
      for (int i = 0; i < n - 1; i++) {
         for (int j = 0; j < n - 1 - i; j++) {
            if (A[j].compareTo(A[j + 1]) > 0) {
```

```
                T temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            }
        }
    }
}

// Selection Sort
public static <T extends Comparable<T>> void selection(T[] A, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (A[j].compareTo(A[min]) < 0) {
                min = j;
            }
        }
        T temp = A[i];
        A[i] = A[min];
        A[minIndex] = temp;
    }
}

// Shell Sort
public static <T extends Comparable<T>> void shell(T[] A, int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            T temp = A[i];
            int j;
            for (j = i; j >= gap && A[j - gap].compareTo(temp) > 0; j -= gap) {
                A[j] = A[j - gap];
            }
            A[j] = temp;
        }
    }
}

// Heap Sort
public static <T extends Comparable<T>> void heap(T[] A, int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(A, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
```

```
            T temp = A[0];
            A[0] = A[i];
            A[i] = temp;

            heapify(A, i, 0);
        }
    }

    private static <T extends Comparable<T>> void heapify(T[] A, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && A[left].compareTo(A[largest]) > 0) {
            largest = left;
        }

        if (right < n && A[right].compareTo(A[largest]) > 0) {
            largest = right;
        }

        if (largest != i) {
            T swap = A[i];
            A[i] = A[largest];
            A[largest] = swap;

            heapify(A, n, largest);
        }
    }

    // Merge Sort
    public static <T extends Comparable<T>> void mergeSort(T[] A) {
        if (A.length < 2) {
            return;
        }

        int mid = A.length / 2;
        T[] left = (T[]) new Comparable[mid];
        T[] right = (T[]) new Comparable[A.length - mid];

        for (int i = 0; i < mid; i++) {
            left[i] = A[i];
        }
        for (int i = mid; i < A.length; i++) {
```

```
        right[i - mid] = A[i];
    }

    mergeSort(left);
    mergeSort(right);
    merge(A, left, right);
}

private static <T extends Comparable<T>> void merge(T[] A, T[] left, T[] right) {
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i].compareTo(right[j]) <= 0) {
            A[k++] = left[i++];
        } else {
            A[k++] = right[j++];
        }
    }
    while (i < left.length) {
        A[k++] = left[i++];
    }
    while (j < right.length) {
        A[k++] = right[j++];
    }
}

// Quick Sort
public static <T extends Comparable<T>> void quick(T[] A, int low, int high) {
    if (low < high) {
        int pi = partition(A, low, high);

        quick(A, low, pi - 1);
        quick(A, pi + 1, high);
    }
}

private static <T extends Comparable<T>> int partition(T[] A, int low, int high) {
    T pivot = A[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (A[j].compareTo(pivot) <= 0) {
            i++;

            T temp = A[i];
```

```
            A[i] = A[j];
            A[j] = temp;
        }
    }

    T temp = A[i + 1];
    A[i + 1] = A[high];
    A[high] = temp;

    return i + 1;
}

public static <T extends Comparable<T>> void quickNR(T[] A, int low, int high) {
    Stack<int[]> stack = new Stack<>();
    stack.push(new int[] { low, high });

    while (!stack.isEmpty()) {
        int[] range = stack.pop();
        low = range[0];
        high = range[1];

        if (low < high) {
            int pi = partition(A, low, high);

            if (pi - 1 > low) {
                stack.push(new int[] { low, pi - 1 });
            }

            if (pi + 1 < high) {
                stack.push(new int[] { pi + 1, high });
            }
        }
    }
}



// Radix Sort
public static void radix(int[] array) {
    int max = Arrays.stream(array).max().getAsInt();

    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortByDigit(array, exp);
    }
```

```
}

//CoutingSort -> RadixSort
private static void countingSortByDigit(int[] array, int exp) {
    int n = array.length;
    int[] output = new int[n];
    int[] count = new int[10];

    for (int i = 0; i < n; i++) {
        int digit = (array[i] / exp) % 10;
        count[digit]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        int digit = (array[i] / exp) % 10;
        output[count[digit] - 1] = array[i];
        count[digit]--;
    }

    System.arraycopy(output, 0, array, 0, n);
}

// Bucket Sort
public static void bucket(float[] array) {
    int n = array.length;
    if (n <= 0) return;

    ArrayList<Float>[] buckets = new ArrayList[n];
    for (int i = 0; i < n; i++) {
        buckets[i] = new ArrayList<>();
    }

    for (float value : array) {
        int bucketIndex = Math.min((int) (value * n), n - 1);
        buckets[bucketIndex].add(value);
    }

    int index = 0;
    for (ArrayList<Float> bucket : buckets) {
      insertion(bucketArray, bucketArray.length);
```

```
        for (float value : bucket) {
            array[index++] = value;
        }
    }
}

// Counting Sort
public static void counting(int[] array) {
    int max = Arrays.stream(array).max().getAsInt();
    int min = Arrays.stream(array).min().getAsInt();
    int range = max - min + 1;

    int[] count = new int[range];
    int[] output = new int[array.length];


    for (int i : array) {
        count[i - min]++;
    }


    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }


    for (int i = array.length - 1; i >= 0; i--) {
        output[count[array[i] - min] - 1] = array[i];
        count[array[i] - min]--;
    }

    System.arraycopy(output, 0, array, 0, array.length);
}


// Merge Sort para int[]
public static void mergeSortInt(int[] A) {
    if (A.length < 2) {
        return;
    }

    int mid = A.length / 2;
    int[] left = new int[mid];
    int[] right = new int[A.length - mid];
```

```
    for (int i = 0; i < mid; i++) {
        left[i] = A[i];
    }
    for (int i = mid; i < A.length; i++) {
        right[i - mid] = A[i];
    }

    mergeSortInt(left);
    mergeSortInt(right);
    mergeInt(A, left, right);
}

//Função de merge para arrays de inteiros
private static void mergeInt(int[] A, int[] left, int[] right) {
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            A[k++] = left[i++];
        } else {
            A[k++] = right[j++];
        }
    }

    while (i < left.length) {
        A[k++] = left[i++];
    }

    while (j < right.length) {
        A[k++] = right[j++];
    }
}

//MergSort com InsertionSort
public static <T extends Comparable<T>> void mergeIns(T[] A, int n) {
    if (A.length < 2) {
        return;
    }


    if (A.length <= n) {
        insertion(A, A.length);
        return;
    }
```

```
        int mid = A.length / 2;
        T[] left = (T[]) new Comparable[mid];
        T[] right = (T[]) new Comparable[A.length - mid];

        for (int i = 0; i < mid; i++) {
            left[i] = A[i];
        }
        for (int i = mid; i < A.length; i++) {
            right[i - mid] = A[i];
        }

        mergeIns(left, n);
        mergeIns(right, n);
        mergeInsArrays(A, left, right);
    }

}
```

Código com os métodos auxiliares(SuportArray.java):

```
package test;

import java.util.Random;

public class SuportArray {
    //Criar diferentes tipos de array, type: 0-Crescente, 1-Decrescente, 2-Elementos
Iguais, 3-Desordenados
    public static Integer[] arrayInt(int n, int type) {
        Integer[] array = new Integer[n];
        Random random = new Random(); /

        switch (type) {
            case 0:
                for (int i = 0; i < n; i++) {
                    array[i] = i;
                }
                break;

            case 1:
                for (int i = 0; i < n; i++) {
                    array[i] = n - 1 - i;
                }
                break;
```

```
            case 2:
                int valorRepetido = random.nextInt(100);
                for (int i = 0; i < n; i++) {
                    array[i] = valorRepetido;
                }
                break;

            case 3: // array aleatório
                for (int i = 0; i < n; i++) {
                    array[i] = random.nextInt(n);
                }
                break;

            default:
            break;
        }

        return array;

    //Mostra todo o array
    public static void mostrarArray(Integer[] array) {
        System.out.print("[");
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i]);
            if (i < array.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

     //Criar diferentes tipos de array(int), type: 0-Crescente, 1-Decrescente, 2-Elementos Iguais, 3-Desordenados
    public static int[] arrayRInt(int n, int type) {
        int[] array = new int[n];
        Random random = new Random();

        switch (type) {
            case 0:
                for (int i = 0; i < n; i++) {
                    array[i] = i;
                }
                break;
```

```
            case 1:
                for (int i = 0; i < n; i++) {
                    array[i] = n - 1 - i;
                }
                break;

            case 2:
                int valorRepetido = random.nextInt(100);
                for (int i = 0; i < n; i++) {
                    array[i] = valorRepetido;
                }
                break;

            case 3: // random array
                for (int i = 0; i < n; i++) {
                    array[i] = random.nextInt(n);
                }
                break;

            default:
                break;
        }

        return array;
    }

     public static void mostrarRArray(int[] array) {
        System.out.print("[");
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i]);
            if (i < array.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }


    //Converte array de Int para um equivalente em Flot, utilizado para testes do
BucketSort
    public static float[] convertToFloatInRange(int[] intArray) {
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
```

```
        for (int num : intArray) {
            if (num < min) {
                min = num;
            }
            if (num > max) {
                max = num;
            }
        }

        float[] floatArray = new float[intArray.length];

        for (int i = 0; i < intArray.length; i++) {
            floatArray[i] = (float) (intArray[i] - min) / (max - min);
        }

        return floatArray;
    }
}
```

Código para teste dos algoritmos de ordenação não lineares(TimeTest.java):

```
package test;

import sort.Sorting;
import java.util.Arrays;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import test.SuportArray;
import java.util.ArrayList;
import java.util.List;

public class TimeTest {
    public static void main(String[] args) {
        // args[0]: quantidade de elementos no array
        // args[1]: quantidade de testes
        // args[2]: tipo de array

        int k = 0;

        List<Long> temposInsertion = new ArrayList<>();
        List<Long> temposBubble = new ArrayList<>();
```

```
        List<Long> temposSelection = new ArrayList<>();
        List<Long> temposShell = new ArrayList<>();
        List<Long> temposHeap = new ArrayList<>();
        List<Long> temposMerge = new ArrayList<>();
        List<Long> temposQuick = new ArrayList<>();


        while (k < Integer.parseInt(args[1])) {
            Integer[] arrayTest = SuportArray.arrayInt(Integer.parseInt(args[0]),
Integer.parseInt(args[2]));

         long t_ini, t_fim;

            // Insertion Sort
            Integer[] arrayInsertion = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.insertion(arrayInsertion, Integer.parseInt(args[0]));
            t_fim = System.nanoTime();
            temposInsertion.add(t_fim - t_ini);

            // Bubble Sort
            Integer[] arrayBubble = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.bubble(arrayBubble, Integer.parseInt(args[0]));
            t_fim = System.nanoTime();
            temposBubble.add(t_fim - t_ini);

            // Selection Sort
            Integer[] arraySelection = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.selection(arraySelection, Integer.parseInt(args[0]));
            t_fim = System.nanoTime();
            temposSelection.add(t_fim - t_ini);

            // Shell Sort
            Integer[] arrayShell = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.shell(arrayShell, Integer.parseInt(args[0]));
            t_fim = System.nanoTime();
            temposShell.add(t_fim - t_ini);

            // Heap Sort
            Integer[] arrayHeap = arrayTest.clone();
            t_ini = System.nanoTime();
```

```
            Sorting.heap(arrayHeap, Integer.parseInt(args[0]));
            t_fim = System.nanoTime();
            temposHeap.add(t_fim - t_ini);

            // Merge Sort
            Integer[] arrayMerge = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.mergeSort(arrayMerge);
            t_fim = System.nanoTime();
            temposMerge.add(t_fim - t_ini);

            // Quick Sort
            Integer[] arrayQuick = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.quick(arrayQuick, 0, Integer.parseInt(args[0]) - 1);
            t_fim = System.nanoTime();
            temposQuick.add(t_fim - t_ini);


            k++;
        }


        String nomeArquivoCSV = "TemposExecucao" + args[0] + "entradas" +
"tipoArray" + args[2] + ".csv";


        try (PrintWriter writer = new PrintWriter(new File(nomeArquivoCSV))) {

            writer.print("Método de Ordenação");
            for (int i = 0; i < Integer.parseInt(args[1]); i++) {
                writer.print(",Teste " + (i + 1));
            }
            writer.println();


            writer.print("Insertion Sort");
            for (Long tempo : temposInsertion) writer.print("," + tempo);
            writer.println();

            writer.print("Bubble Sort");
            for (Long tempo : temposBubble) writer.print("," + tempo);
            writer.println();
```

```
            writer.print("Selection Sort");
            for (Long tempo : temposSelection) writer.print("," + tempo);
            writer.println();

            writer.print("Shell Sort");
            for (Long tempo : temposShell) writer.print("," + tempo);
            writer.println();

            writer.print("Heap Sort");
            for (Long tempo : temposHeap) writer.print("," + tempo);
            writer.println();

            writer.print("Merge Sort");
            for (Long tempo : temposMerge) writer.print("," + tempo);
            writer.println();

            writer.print("Quick Sort");
            for (Long tempo : temposQuick) writer.print("," + tempo);
            writer.println();

        } catch (FileNotFoundException e) {
            System.err.println("Erro ao criar o arquivo CSV: " + e.getMessage());
        }
    }
}
```

Código para teste dos algoritmos de ordenação não lineares(TimeTestlinear.java):

```
package test;

import sort.Sorting;
import java.util.Arrays;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import test.SuportArray;
import java.util.ArrayList;
import java.util.List;

public class TimeTestLinear {
    public static void main(String[] args) {
        // args[0]: quantidade de elementos no array
        // args[1]: quantidade de testes
        // args[2]: tipo de array
```

```
        int k = 0; // contador para os testes

        List<Long> temposCounting = new ArrayList<>();
        List<Long> temposRadix = new ArrayList<>();
        List<Long> temposBucket = new ArrayList<>();

        while (k < Integer.parseInt(args[1])) {
            int[] arrayTest = SuportArray.arrayRInt(Integer.parseInt(args[0]),
Integer.parseInt(args[2]));
            float[] arrayBucket =  SuportArray.convertToFloatInRange(arrayTest);

         long t_ini, t_fim;

            // Couting Sort
            int[] arrayCounting = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.counting(arrayCounting);
            t_fim = System.nanoTime();
            temposCounting.add(t_fim - t_ini);

            // Radix Sort
            int[] arrayRadix = arrayTest.clone();
            t_ini = System.nanoTime();
            Sorting.radix(arrayRadix);
            t_fim = System.nanoTime();
            temposRadix.add(t_fim - t_ini);

            // Selection Bucket
            t_ini = System.nanoTime();
            Sorting.bucket(arrayBucket);
            t_fim = System.nanoTime();
            temposBucket.add(t_fim - t_ini);


            k++;
        }


        String nomeArquivoCSV = "TemposExecucaoLinear" + args[0] + "entradas" +
"tipoArray" + args[2] + ".csv";


        try (PrintWriter writer = new PrintWriter(new File(nomeArquivoCSV))) {
```

```
            writer.print("Método de Ordenação");
            for (int i = 0; i < Integer.parseInt(args[1]); i++) {
                writer.print(",Teste " + (i + 1));
            }
            writer.println();


            writer.print("Counting Sort");
            for (Long tempo : temposCounting) writer.print("," + tempo);
            writer.println();

            writer.print("Radix Sort");
            for (Long tempo : temposRadix) writer.print("," + tempo);
            writer.println();

            writer.print("Bucket Sort");
            for (Long tempo : temposBucket) writer.print("," + tempo);
            writer.println();

        } catch (FileNotFoundException e) {
            System.err.println("Erro ao criar o arquivo CSV: " + e.getMessage());
        }
    }
}
```

Código para geração do gráfico de linha de todos os algoritmos(graphAll.py):

```
import matplotlib.pyplot as plt
import pandas as pd


dados = pd.read_csv('Tempos/Tempos Formatados/Tempos Totais (desordenado).csv')

tamanhos = ['10 entradas', '100 entradas', '1000 entradas', '10000 entradas', '100000
entradas', '1000000 entradas']

dados[tamanhos] = dados[tamanhos].replace({',': '.'}, regex=True)
dados[tamanhos] = dados[tamanhos].apply(pd.to_numeric)

tamanhos_int = [10, 100, 1000, 10000, 100000, 1000000]

cores = ['tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple',
        '#A52A2A', '#FFC0CB', '#808080', '#6B8E23', '#00FFFF', '#00FF00']
```

```
metodos = [
    ('Insertion Sort', cores[0]),
    ('Bubble Sort', cores[1]),
    ('Selection Sort', cores[2]),
    ('Shell Sort', cores[3]),
    ('Heap Sort', cores[4]),
    ('Merge Sort', cores[5]),
    ('Quick Sem Recurssão', cores[7]),
    ('Counting Sort', cores[8]),
    ('Radix Sort', cores[9]),
    ('Bucket Sort', cores[10])
]

plt.figure(figsize=(12, 8))

for metodo, cor in metodos:
    tempos = dados.loc[dados['Métodos de Ordenação'] == metodo,
tamanhos].values.flatten()
    plt.plot(tamanhos_int, tempos, label=metodo, marker='o', color=cor, linestyle='-',
linewidth=2)



plt.yscale('log')
plt.xscale('log')

plt.xlabel('Tamanho da Entrada (log scale)')
plt.ylabel('Tempo de Execução (log scale)')
plt.title('Comparação de Tempo de Execução para Diferentes Métodos de Ordenação')

plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)

plt.tight_layout()
plt.savefig("comparacao_metodos_ordenacao_all.png")
plt.close()
```

Código para geração do gráfico de linha de todos os algoritmos (graphColunas.py):

```
import matplotlib.pyplot as plt
import pandas as pd
```

```
dados = pd.read_csv('Tempos/Tempos Formatados/Tempos Totais (desordenado).csv')


metodos = ['Insertion Sort', 'Bubble Sort', 'Selection Sort', 'Shell Sort', 'Heap Sort',
        'Merge Sort', 'Quick Sort', 'Quick Sem Recurssão', 'Counting Sort',
        'Radix Sort', 'Bucket Sort']

tamanhos = ['10 entradas', '100 entradas', '1000 entradas', '10000 entradas',
         '100000 entradas', '1000000 entradas']


for tamanho in tamanhos:
   dados[tamanho] = dados[tamanho].str.replace(',', '.').astype(float)

def salvar_grafico(tamanho):
   plt.figure(figsize=(12, 6))


   tempos = []
   for metodo in metodos:
      tempo = dados.loc[dados['Métodos de Ordenação'] == metodo,
tamanho].values.flatten()
      tempos.append(tempo[0])

   cores = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
         '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf', '#9eeb75']  # Hexadecimal


   plt.bar(metodos, tempos, color=cores[:len(metodos)], width=0.5)

   plt.yscale('log')


   plt.xlabel('Métodos de Ordenação')
   plt.ylabel('Tempo de Execução (log scale)')
   plt.title(f'Tempo de Execução para {tamanho}')


   plt.grid(True, which="both", linestyle="--", linewidth=0.5)

   plt.xticks(rotation=45, ha='right')

   plt.tight_layout()
```

```
    nome_arquivo = f'{tamanho}.png'
    plt.savefig(nome_arquivo)
    plt.close()

for tamanho in tamanhos:
    salvar_grafico(tamanho)
```

Código para geração do gráfico de comparação dos algoritmos MergeSort (graphMerge.py):

```
import matplotlib.pyplot as plt
import pandas as pd

dados = pd.read_csv('merge/merges.csv')

tamanhos = ['10 entradas', '100 entradas', '1000 entradas', '10000 entradas']

for tamanho in tamanhos:

    dados[tamanho] = dados[tamanho].astype(str).str.replace(',', '.').astype(float)


metodos = [
    'Merge Comparable',
    'Merge Int',
    'Merge Insertion 5',
    'Merge Insertion 10',
    'Merge Insertion 50'
]


tamanhos_int = [10, 100, 1000, 10000]


plt.figure(figsize=(12, 8))

cores = ['b', 'g', 'r', 'c', 'm']

for metodo, cor in zip(metodos, cores):
    tempos = dados.loc[dados['Métodos de Ordenação'] == metodo,
tamanhos].values.flatten()
```

```
    plt.plot(tamanhos_int, tempos, label=metodo, marker='o', color=cor, linestyle='-',
linewidth=2)


plt.yscale('log')


plt.xlabel('Tamanho da Entrada')
plt.ylabel('Tempo de Execução (log scale)')
plt.title('Comparação de Métodos de Ordenação Merge')


plt.legend()


plt.grid(True, which="both", linestyle="--", linewidth=0.5)


plt.tight_layout()


plt.savefig('comparacao_metodos_merge.png', format='png')

plt.close()
```

Código para geração do gráfico de comparação dos algoritmos com entradas cresecentes, decrescentes e repetidas (graphOrd.py):

```
import matplotlib.pyplot as plt
import pandas as pd

dados = pd.read_csv('Tempos/Tempos Formatados/Tempos Totais Ord.csv')


metodos = ['Insertion Sort', 'Bubble Sort', 'Selection Sort', 'Shell Sort', 'Heap Sort',
        'Merge Sort', 'Quick Sort', 'Quick Sem Recurssão', 'Counting Sort',
        'Radix Sort', 'Bucket Sort']


tamanhos = ['Crescente', 'Decrescente', 'Iguais'
]


for tamanho in tamanhos:
```

```
    dados[tamanho] = dados[tamanho].str.replace(',', '.').astype(float)

def salvar_grafico(tamanho):
    plt.figure(figsize=(12, 6))


    tempos = []
    for metodo in metodos:
        tempo = dados.loc[dados['Métodos de Ordenação'] == metodo,
tamanho].values.flatten()
        tempos.append(tempo[0])


    cores = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
             '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf', '#9eeb75']

    plt.bar(metodos, tempos, color=cores[:len(metodos)], width=0.5)


    plt.yscale('log')


    plt.xlabel('Métodos de Ordenação')
    plt.ylabel('Tempo de Execução (log scale)')
    plt.title(f'Tempo de Execução para {tamanho}')


    plt.grid(True, which="both", linestyle="--", linewidth=0.5)


    plt.xticks(rotation=45, ha='right')

    # Ajusta automaticamente o layout
    plt.tight_layout()


    nome_arquivo = f'{tamanho}.png'
    plt.savefig(nome_arquivo)
    plt.close()

for tamanho in tamanhos:
    salvar_grafico(tamanho)
```

Código para geração do gráfico de comparação dos algoritmos QuickSort (graphQuick.py):

```
import matplotlib.pyplot as plt
import pandas as pd

dados = pd.read_csv('Tempos/Tempos Formatados/Tempos Totais (desordenado).csv')

tamanhos = ['10 entradas', '100 entradas', '1000 entradas', '10000 entradas']

for tamanho in tamanhos:
    dados[tamanho] = dados[tamanho].str.replace(',', '.').astype(float)

quick_sort_tempos = dados.loc[dados['Métodos de Ordenação'] == 'Quick Sort',
tamanhos].values.flatten()
quick_sem_recurssao_tempos = dados.loc[dados['Métodos de Ordenação'] == 'Quick
Sem Recurssão', tamanhos].values.flatten()

tamanhos_int = [10, 100, 1000, 10000]

plt.figure(figsize=(10, 6))

plt.plot(tamanhos_int, quick_sort_tempos, label='Quick Sort', marker='o', color='b',
linestyle='-', linewidth=2)

plt.plot(tamanhos_int, quick_sem_recurssao_tempos, label='Quick Sem Recursão',
marker='o', color='r', linestyle='-', linewidth=2)

plt.yscale('log')


plt.xlabel('Tamanho da Entrada')
plt.ylabel('Tempo de Execução (log scale)')
plt.title('Comparação de Quick Sort e Quick Sem Recursão')

plt.legend()

plt.grid(True, which="both", linestyle="--", linewidth=0.5)

plt.tight_layout()

plt.savefig('comparacao_quick_sort_quick_sem_recurssao.png', format='png')

plt.close()
```