UNIVERSIDADE ESTADUAL DO PIAUÍ - UESPI
TERESINA, 28 DE NOVEMBRO DE 2025
DISCIPLINA: PROJETO E ANÁLISE DE ALGORITMO
PROF°: MARCUS VINÍCIUS
ALUNO: JOÃO ANTONIO GONÇALVES LAVRAS, RAFAEL DE MOURA PAZ - BLOCO 4

RELATÓRIO DE ANÁLISE DE ALGORITMOS DE ORDENAÇÃO - AVALIAÇÃO 3

TERESINA - PIAUÍ

1. INTRODUÇÃO

A ordenação de dados é um processo fundamental na ciência da computação e influencia diretamente a eficiência de diversos sistemas de processamento de informação. Este trabalho tem como objetivo analisar comparativamente o desempenho de algoritmos de ordenação elementares — *Bubble Sort*, *Insertion Sort* e *Selection Sort* — e algoritmos considerados mais eficientes, como *QuickSort*, *MergeSort* e *HeapSort*, a fim de compreender suas diferenças estruturais e operacionais.

Para isso, foi desenvolvido um programa em Java capaz de manipular vetores de tamanhos progressivos (10, 100, 1.000, 10.000 e 100.000), compostos por valores inteiros únicos. Os testes foram executados em diferentes cenários de entrada — ordenada, quase ordenada, inversamente ordenada e aleatória — permitindo observar como a teoria da complexidade computacional se manifesta na prática e evidenciando as vantagens e limitações de cada abordagem de ordenação.

2. IMPLEMENTAÇÃO

A implementação do projeto foi realizada em linguagem Java, utilizando exclusivamente vetores (arrays) como estrutura de dados para armazenar, manipular e ordenar as sequências numéricas. O programa foi desenvolvido de forma modular, separando os algoritmos de ordenação em classes ou funções independentes, o que facilita a manutenção e a compreensão do código.

O trabalho utiliza vetores (int[]) como estrutura primária para representar as sequências de elementos a serem ordenadas. Optamos por arrays por serem simples, eficientes em acesso por índice e por corresponderem diretamente ao enunciado (arranjos). Para evitar mutação do vetor original durante os testes, cada algoritmo opera sobre uma cópia do vetor de entrada (método utilitário copiarVetor).

Além dos vetores, o sistema emprega objetos de suporte:

- **Resultado** — objeto que mantém contadores de *comparações* e *atribuições* durante a execução de um algoritmo, e métodos para incrementá-los e obtê-los.
- **RegistroCSV** — responsável por criar e preencher o arquivo resultados.csv com as colunas: Algoritmo, Tamanho, Cenario, Tempo(ns), Tempo(ms), Comparacoes, Atribuicoes, VetorAntes, VetorDepois.

- **VetorUtils** — gerador de vetores (ordenado, inverso, quase ordenado, aleatório), cópia e utilitários para converção em string.

2.1 Estrutura do Projeto

A organização do projeto foi estruturada seguindo uma arquitetura clara, modular e de fácil manutenção. A pasta principal, **Analise_De_Algoritmos**, foi dividida em três blocos fundamentais: *Domain*, *Utils* e *Test*. No diretório **Domain**, concentram-se todas as implementações dos algoritmos de ordenação utilizados no estudo, incluindo Bubble Sort, Insertion Sort, Selection Sort, QuickSort, MergeSort e HeapSort, além da classe **Resultado**, responsável por armazenar as métricas coletadas durante cada execução. Essa separação permite que os algoritmos funcionem de maneira independente, favorecendo a reutilização e facilitando a leitura do código.

O diretório **Utils** agrupa as classes auxiliares responsáveis por sustentar o funcionamento geral do sistema. Nele estão a classe **VetorUtils**, que gera os vetores utilizados nos experimentos, o enum **Cenario**, que padroniza os tipos de entrada avaliados, e a classe **RegistroCSV**, que registra os resultados no arquivo final. Há também a classe **TempoUtils**, usada opcionalmente para conversões e formatações de tempo. Por fim, o diretório **Test** contém a classe **Application**, ponto de entrada do programa, responsável por coordenar o fluxo completo dos testes. Essa organização modular reforça a clareza da implementação e favorece a expansão futura do projeto.

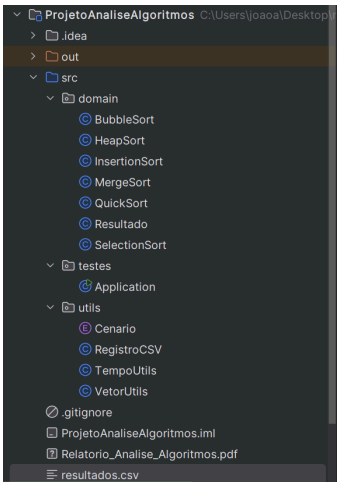

Figura 1 - Organização dos Códigos
Fonte: Adaptado pelo Autor

2.2 Fluxo do Programa (Visão geral)

O fluxo geral do programa foi estruturado para garantir que todos os algoritmos de ordenação fossem testados de maneira padronizada e comparável. Inicialmente, o software percorre cada cenário definido — *ordenado*, *inversamente ordenado*, *aleatório* e *quase ordenado*. Em seguida, para cada um desses cenários, são avaliados diferentes tamanhos de vetores (10, 100, 1.000, 10.000 e 100.000 elementos). Para cada combinação de cenário e tamanho, um vetor original é gerado utilizando a classe **VetorUtils**, garantindo que todos os algoritmos atuem sobre a mesma base de teste. Esse vetor é então copiado antes da execução de cada algoritmo, assegurando que nenhuma execução afete a próxima.

Após preparar o vetor, o programa processa cada um dos seis algoritmos avaliados: Bubble Sort, Insertion Sort, Selection Sort, QuickSort, MergeSort e HeapSort. Para cada execução, cria-se um objeto **Resultado**, que armazena métricas como comparações e atribuições. O tempo é medido com alta precisão utilizando System.nanoTime(). O algoritmo instrumentado é então executado, atualizando automaticamente essas métricas durante sua operação. Ao final da ordenação, o programa registra no arquivo CSV todas as informações relevantes, incluindo o vetor antes da ordenação, o vetor final, o tempo consumido em nanosegundos e milissegundos, além do total de comparações e atribuições. O fluxo completo pode ser resumido no diagrama textual: **Main → cenários → tamanhos → geração do vetor → execução dos algoritmos → registro dos resultados no CSV**.

2.3 Formato de entrada e saída

A entrada do programa não depende de arquivos externos, pois todos os vetores utilizados nos testes são gerados automaticamente pela própria aplicação. Os parâmetros necessários — como tamanhos dos vetores e cenários de teste — são definidos internamente no código, garantindo controle total sobre o ambiente de execução. Essa abordagem elimina interferências externas e padroniza os experimentos, permitindo comparar algoritmos em condições idênticas.

A saída principal produzida pelo sistema é o arquivo **resultados.csv**, que registra detalhadamente as métricas de desempenho de cada algoritmo. Esse arquivo contém colunas

como: *Algoritmo, Tamanho, Cenário, Tempo(ns), Tempo(ms), Comparações, Atribuições, VetorAntes,* e *VetorDepois*. Posteriormente, esse CSV é utilizado como entrada por um script em Python que gera o relatório final em PDF, contendo tabelas, análises e gráficos comparativos.

2.4 Controles de experimentação e decisões

Para garantir consistência nos testes, todos os vetores utilizados são compostos por valores inteiros únicos, evitando duplicidades que poderiam influenciar o comportamento dos algoritmos. A medição de tempo é realizada em nanosegundos para maior precisão, sendo posteriormente convertida para milissegundos no relatório para facilitar a leitura. Além disso, cada algoritmo acumula duas métricas importantes: número de comparações realizadas e número de atribuições feitas no vetor, o que permite uma análise mais aprofundada do custo computacional além do tempo de execução.

2.5 Códigos Essenciais Utilizados e suas fundamentações

A pasta **Domain/** contém exclusivamente os algoritmos de ordenação instrumentados, cada um implementado em sua própria classe: *BubbleSort.java*, *InsertionSort.java*, *SelectionSort.java*, *QuickSort.java*, *MergeSort.java* e *HeapSort.java*. Também há a classe *Resultado.java*, responsável por armazenar métricas como tempo, comparações, atribuições e vetores antes e depois da ordenação. Essa separação garante independência entre a lógica dos algoritmos e o restante do sistema.

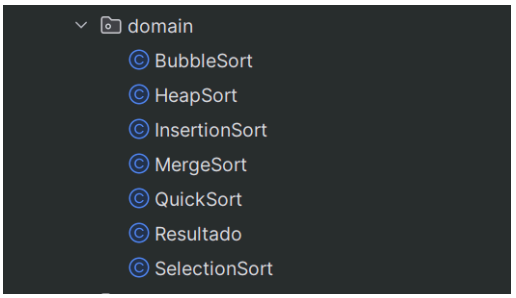

Figura 2 - Estruturação da pasta domain
Fonte: Adaptado pelo Autor

2.5.1. BubbleSort.java

O código do *BubbleSort* implementa a versão clássica do algoritmo, percorrendo o vetor repetidamente e comparando elementos adjacentes, realizando trocas sempre que necessário para empurrar os valores maiores para o final da lista. Cada comparação entre arr[j] e arr[j+1] é registrada no objeto Resultado, assim como cada atribuição envolvida nas trocas, permitindo medir precisamente o custo operacional do método. Trata-se de um algoritmo **O(n²)**, simples porém ineficiente para grandes conjuntos de dados, sendo útil neste projeto justamente como referência de desempenho baixo.

Implementação:

```
package domain;

public class BubbleSort {

    public void bubbleSort(int[] arr, Resultado r) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {

                r.addComparacao(); // comparação arr[j] > arr[j+1]

                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    r.addAtribuicao();

                    arr[j] = arr[j + 1];
                    r.addAtribuicao();

                    arr[j + 1] = temp;
                    r.addAtribuicao();
                }
            }
        }
    }
}
```

2.5.2. HeapSort.java

O *HeapSort* utiliza uma estrutura implícita de heap máximo dentro do próprio vetor para ordenar os elementos de forma eficiente. Primeiramente, o método constrói o heap chamando heapify de forma decrescente, garantindo que o maior elemento esteja sempre na raiz; em seguida, realiza a extração sucessiva dessa raiz, trocando-a com a última posição do heap

reduzido. Cada comparação e atribuição é contabilizada no objeto Resultado, permitindo observar a complexidade **O(n log n)** do algoritmo. A função auxiliar heapify mantém a propriedade do heap de forma recursiva, ajustando as posições sempre que necessário.

Implementação:

```
package domain;

public class HeapSort {

    public void heapSort(int[] arr, Resultado r) {
        int n = arr.length;

        // construir heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i, r);
        }

        // extrair elementos
        for (int i = n - 1; i > 0; i--) {

            int temp = arr[0];
            r.addAtribuicao();

            arr[0] = arr[i];
            r.addAtribuicao();

            arr[i] = temp;
            r.addAtribuicao();

            heapify(arr, i, 0, r);
        }
    }

    private void heapify(int[] arr, int n, int i, Resultado r) {
        int maior = i;
        int esq = 2 * i + 1;
        int dir = 2 * i + 2;

        // compara esquerda
        if (esq < n) {
            r.addComparacao();
            if (arr[esq] > arr[maior]) {
                maior = esq;
            }
        }

        // compara direita
        if (dir < n) {
            r.addComparacao();
            if (arr[dir] > arr[maior]) {
                maior = dir;
            }
```

```
        }

        // troca se necessário
        if (maior != i) {
            int temp = arr[i];
            r.addAtribuicao();

            arr[i] = arr[maior];
            r.addAtribuicao();

            arr[maior] = temp;
            r.addAtribuicao();

            heapify(arr, n, maior, r);
        }
    }
}
```

2.5.3. InsertionSort.java

O *InsertionSort* percorre o vetor a partir do segundo elemento, selecionando um valor e deslocando todos os elementos maiores que ele uma posição à frente até encontrar sua posição correta. Durante o processo são registradas comparações dentro do laço while e as atribuições correspondentes às movimentações. Sua complexidade é **O(n²)** no pior caso e **O(n)** no melhor caso (vetor já ordenado), tornando-o ideal para pequenas entradas ou entradas quase ordenadas, o que o torna especialmente interessante nos cenários deste trabalho.

Implementação:

```
package domain;

public class InsertionSort {

    public void insertionSort(int[] arr, Resultado r) {
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int currentValue = arr[i];
            r.addAtribuicao();

            int j = i - 1;

            while (j >= 0) {
                r.addComparacao(); // comparação arr[j] >
currentValue

                if (arr[j] > currentValue) {
                    arr[j + 1] = arr[j];
```

```
                    r.addAtribuicao();

                    j--;
                } else {
                    break;
                }
            }

            arr[j + 1] = currentValue;
            r.addAtribuicao();
        }
    }
}
```

2.5.4. MergeSort.java

O *MergeSort* implementa a estratégia de divisão e conquista, dividindo o vetor repetidamente ao meio até atingir subvetores unitários e, em seguida, mesclando essas partes de forma ordenada. A etapa merge realiza as comparações entre os subvetores auxiliares L e R, transferindo para o vetor original o menor elemento a cada passo e contabilizando cada comparação e atribuição. Apesar de exigir memória adicional, o algoritmo garante complexidade **O(n log n)** em todos os cenários, sendo um dos métodos mais estáveis e previsíveis da análise.

Implementação:

```
package domain;

public class MergeSort {

    public void mergeSort(int[] arr, int inicio, int fim,
Resultado r) {
        if (inicio < fim) {
            int meio = (inicio + fim) / 2;

            mergeSort(arr, inicio, meio, r);
            mergeSort(arr, meio + 1, fim, r);

            merge(arr, inicio, meio, fim, r);
        }
    }

    private void merge(int[] arr, int inicio, int meio, int fim,
Resultado r) {

        int n1 = meio - inicio + 1;
        int n2 = fim - meio;

        int[] L = new int[n1];
```

```
        int[] R = new int[n2];

        // copia
        for (int i = 0; i < n1; i++) {
            L[i] = arr[inicio + i];
            r.addAtribuicao();
        }
        for (int j = 0; j < n2; j++) {
            R[j] = arr[meio + 1 + j];
            r.addAtribuicao();
        }

        int i = 0, j = 0, k = inicio;

        while (i < n1 && j < n2) {

            r.addComparacao(); // L[i] <= R[j]

            if (L[i] <= R[j]) {
                arr[k] = L[i];
                r.addAtribuicao();
                i++;
            } else {
                arr[k] = R[j];
                r.addAtribuicao();
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            r.addAtribuicao();
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            r.addAtribuicao();
            j++;
            k++;
        }
    }
}
```

2.5.5. QuickSort.java

O *QuickSort* utiliza a técnica de particionamento baseada em um pivô, escolhido aleatoriamente para reduzir a probabilidade de casos degenerados. O método particiona reorganiza os elementos menores ou iguais ao pivô à esquerda e os maiores à direita, registrando comparações e trocas através do método swap. Após particionar, o algoritmo é

aplicado recursivamente às subpartes, resultando em um desempenho médio de **O(n log n)**, embora podendo atingir **O(n²)** em particionamentos ruins. É o algoritmo mais rápido na prática para vetores grandes e aleatórios.

Implementação:

```
package domain;

import java.util.Random;

public class QuickSort {

    private Random rand = new Random();

    public void quickSort(int[] arr, int inicio, int fim,
Resultado r) {
        if (inicio < fim) {
            int p = particiona(arr, inicio, fim, r);
            quickSort(arr, inicio, p - 1, r);
            quickSort(arr, p + 1, fim, r);
        }
    }

    private int particiona(int[] arr, int inicio, int fim,
Resultado r) {
        // Escolher pivot aleatório
        int pivotIndex = inicio + rand.nextInt(fim - inicio + 1);
        swap(arr, pivotIndex, fim, r);

        int pivo = arr[fim];
        r.addAtribuicao();

        int i = inicio - 1;

        for (int j = inicio; j < fim; j++) {
            r.addComparacao(); // arr[j] <= pivo
            if (arr[j] <= pivo) {
                i++;
                swap(arr, i, j, r);
            }
        }

        swap(arr, i + 1, fim, r);
        return i + 1;
    }

    private void swap(int[] arr, int i, int j, Resultado r) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        r.addAtribuicao();
```

```
        r.addAtribuicao();
        r.addAtribuicao();
    }
}
```

2.5.6. Resultado.java

A classe *Resultado* funciona como uma estrutura auxiliar que contabiliza operações internas dos algoritmos, registrando o número total de comparações e atribuições. Ela encapsula essas métricas e fornece métodos de incremento (addComparacao, addAtribuicao) e consulta, permitindo que cada execução seja avaliada quantitativamente. Esse monitoramento é essencial para comparar algoritmos além do tempo de execução, fornecendo indicadores diretos do custo operacional de cada estratégia.

Implementação:

```
package domain;

public class Resultado {

    private long comparacoes;
    private long atribuicoes;

    public Resultado() {
        this.comparacoes = 0;
        this.atribuicoes = 0;
    }

    public void addComparacao() {
        this.comparacoes++;
    }

    public void addAtribuicao() {
        this.atribuicoes++;
    }

    public long getComparacoes() {
        return comparacoes;
    }

    public long getAtribuicoes() {
        return atribuicoes;
    }

    public void reset() {
```

```
        this.comparacoes = 0;
        this.atribuicoes = 0;
    }
}
```

2.5.7. SelectionSort.java

O *SelectionSort* percorre o vetor identificando o menor elemento da parte não ordenada e trocando-o com a posição inicial dessa região. Para cada elemento examinado, uma comparação é registrada e, ao final de cada varredura, as três atribuições envolvidas na troca são contabilizadas. Embora simples, o método realiza sempre o mesmo número de comparações (**$O(n^2)$**), independentemente da ordem inicial do vetor, o que o torna útil para observar comportamentos previsíveis de custo fixo.

Implementação:

```
package domain;

public class SelectionSort {

    public void selectionSort(int[] arr, Resultado r) {

        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {

            int minIndex = i;

            for (int j = i + 1; j < n; j++) {

                r.addComparacao(); // arr[j] < arr[minIndex]

                if (arr[j] < arr[minIndex]) {

                    minIndex = j;

                }

            }

            int temp = arr[i];

            r.addAtribuicao();

            arr[i] = arr[minIndex];

            r.addAtribuicao();

            arr[minIndex] = temp;
```

```
            r.addAtribuicao();
        }
    }
}
```

2.5.7. VetorUtils.java

Na pasta Utils/ estão os componentes de suporte utilizados em toda a aplicação. VetorUtils.java é responsável pela geração dos vetores de entrada conforme o cenário; Cenario.java define os tipos de entrada possíveis (ORDENADO, INVERSO, ALEATORIO e QUASE_ORDENADO); RegistroCSV.java gerencia a escrita dos resultados no arquivo resultados.csv; e TempoUtils.java pode ser utilizado para padronização das medições de tempo. Esses utilitários permitem que o código principal permaneça limpo e focado na lógica experimental. Como destaque principal da criação dos cenários, foi utilizado a seguinte implementação do VetorUtils:

```
package utils;
import java.util.Random;
public class VetorUtils {
    // Gera vetor único aleatório de tamanho n
    public static int[] gerarVetorAleatorio(int n) {
        int[] vetor = new int[n];
        for (int i = 0; i < n; i++) {
            vetor[i] = i; // valores únicos
        }
        embaralhar(vetor);
        return vetor;
    }

    // Gera vetor ordenado
    public static int[] gerarVetorOrdenado(int n) {
```

```
        int[] vetor = new int[n];
        for (int i = 0; i < n; i++) vetor[i] = i;
        return vetor;
    }

    // Gera vetor inversamente ordenado
    public static int[] gerarVetorInverso(int n) {
        int[] vetor = new int[n];
        for (int i = 0; i < n; i++) vetor[i] = n - i - 1;
        return vetor;
    }

        // Gera vetor quase ordenado (90% ordenado, 10% embaralhado)
    public static int[] gerarVetorQuaseOrdenado(int n) {
        int[] vetor = gerarVetorOrdenado(n);
        Random rand = new Random();
        int numTrocas = n / 10;
        for (int i = 0; i < numTrocas; i++) {
            int a = rand.nextInt(n);
            int b = rand.nextInt(n);
            int temp = vetor[a];
            vetor[a] = vetor[b];
            vetor[b] = temp;
        }
        return vetor;
    }
```

```
    // Copia vetor para não alterar o original
    public static int[] copiarVetor(int[] vetor) {
        int[] copia = new int[vetor.length];
        System.arraycopy(vetor, 0, copia, 0, vetor.length);
        return copia;
    }
    // Método Fisher-Yates para embaralhar
    private static void embaralhar(int[] vetor) {
        Random rand = new Random();
        for (int i = vetor.length - 1; i > 0; i--) {
            int j = rand.nextInt(i + 1);
            int temp = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = temp;
        }
    }

         // Converte vetor para string formatada (ex: [1, 5, 2,
8])
    public static String vetorParaString(int[] vetor) {
        if (vetor == null) return "[]";
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < vetor.length; i++) {
            sb.append(vetor[i]);
            if (i < vetor.length - 1) sb.append(", ");
        }
        sb.append("]");
```

```
        return sb.toString();
    }
}
```

Por fim, a pasta Test/ abriga o arquivo Application.java, que centraliza a execução de todos os testes. É nesse arquivo que ocorre o fluxo completo: iteração pelos cenários, tamanhos e algoritmos; geração dos vetores; execução dos algoritmos instrumentados; captura das métricas; e gravação dos resultados. Trata-se do ponto de entrada da aplicação e da rotina responsável por produzir o CSV final utilizado posteriormente pelo script Python de análise.

3. ANÁLISE DE COMPLEXIDADE

Os algoritmos implementados apresentam comportamentos distintos em termos de complexidade computacional, tanto no melhor quanto no pior caso. O Bubble Sort e o Insertion Sort possuem melhor caso O(n) quando o vetor está ordenado, mas seu pior caso é $O(n^2)$. Já o Selection Sort mantém complexidade $O(n^2)$ em todos os cenários, pois sempre percorre o vetor inteiro independentemente da ordem inicial. Os algoritmos mais eficientes — Merge Sort, Quick Sort e Heap Sort — possuem desempenho teórico superior: o Merge Sort opera em O(n log n) em qualquer situação, o Heap Sort também possui desempenho O(n log n) tanto no melhor quanto no pior caso, enquanto o Quick Sort possui melhor e médio caso O(n log n), mas seu pior caso é $O(n^2)$ quando o pivô escolhido é desfavorável.

Observando a natureza das operações de cada algoritmo, percebe-se que aqueles baseados em comparação simples e trocas diretas, como Bubble Sort e Insertion Sort, são fortemente impactados pela estrutura inicial do vetor, resultando em crescimento quadrático na maioria das situações. O Selection Sort, apesar de estável em complexidade, é ineficiente na prática por sempre realizar o mesmo número de comparações, mesmo em vetores quase ordenados. Por outro lado, algoritmos com divisão e conquista — como Merge Sort e Quick Sort — reduzem significativamente o número de operações ao particionar o problema em subproblemas menores, o que explica seu desempenho superior. O Heap Sort, apoiado na estrutura de heap binário, também mantém ordem logarítmica nas operações essenciais, garantindo consistência e eficiência mesmo em dados originalmente desordenados.

Quando comparados entre si, os algoritmos quadráticos (**Bubble, Selection e Insertion Sort**) tornam-se inviáveis para vetores grandes, enquanto os algoritmos de complexidade **O(n log n)** se destacam pela escalabilidade. O Quick Sort tende a ser o mais rápido na prática devido

à boa localidade de memória e ao particionamento eficiente, embora dependa de escolhas adequadas de pivô; o Merge Sort é o mais estável em termos de tempo, mas exige memória adicional; e o Heap Sort, embora ligeiramente mais lento que o Quick Sort em média, oferece forte previsibilidade de desempenho. Assim, fica claro que, para grandes volumes de dados, algoritmos de divisão e conquista ou baseados em estruturas de heap são amplamente superiores às abordagens quadráticas tradicionais.

4. LISTAGEM DOS TESTES EXECUTADOS

A seguir, apresentamos a listagem dos testes realizados para avaliar o desempenho dos algoritmos de ordenação implementados. Esses testes foram conduzidos utilizando diferentes tamanhos de entrada e cenários distintos de vetores, permitindo observar como cada algoritmo reage a variações na organização inicial dos dados. A análise dos resultados possibilita comparar tempo de execução, número de comparações, atribuições e comportamento geral dos métodos, oferecendo uma visão clara e fundamentada sobre a eficiência prática de cada abordagem.

4.1. CENÁRIO ORDENADO

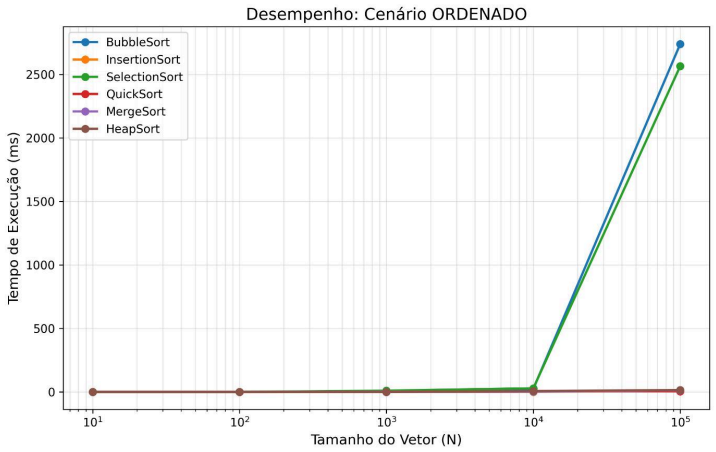

Figura 3 - Gráfico do Cenário Ordenado
Fonte: Adaptado pelo Autor. Gerado no Python.

A análise visual do gráfico (Figura 3) no cenário Ordenado revela uma dicotomia acentuada no comportamento assintótico dos algoritmos conforme o vetor (N) cresce. Observa-se que o Bubble Sort (linha azul) e o Selection Sort (linha verde) apresentam um crescimento abrupto no tempo de execução, ultrapassando a marca de 2.500ms para $N=10^5$. Isso evidencia

graficamente a ineficiência de suas abordagens quadráticas ($O(n^2)$) para grandes volumes de dados, mesmo em um cenário teoricamente favorável para comparações. Em contraste, forma-se um agrupamento de alta eficiência próximo ao eixo horizontal, onde Insertion Sort, QuickSort, MergeSort e HeapSort mantêm tempos de execução desprezíveis (próximos a 0ms). Destaca-se o Insertion Sort, cujo comportamento linear (O(n)) no melhor caso é confirmado pela curva plana, e o QuickSort, que, ao manter-se estável junto aos algoritmos de complexidade log-linear (O(n log n)), demonstra que a escolha do pivô foi adequada, evitando a degradação para o pior caso quadrático típico de implementações ingênuas em arrays ordenados. Por fim, a estabilidade do MergeSort e HeapSort, indistinguíveis visualmente do eixo X nesta escala, confirma a robustez de suas estruturas, validando a superioridade prática dos métodos avançados (e do Insertion Sort neste caso específico) sobre os elementares, cuja divergência de desempenho se torna proibitiva à medida que N atinge ordens de magnitude superiores.

4.1. CENÁRIO INVERSO

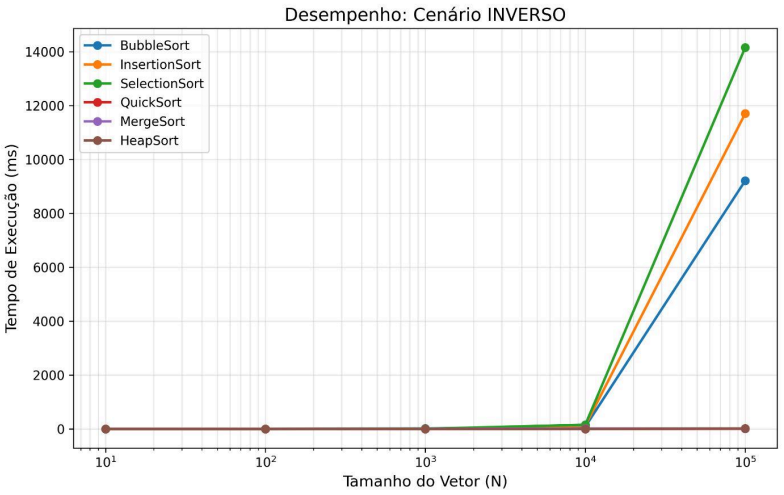

Figura 4 - Gráfico do Cenário Inverso
Fonte: Adaptado pelo Autor. Gerado no Python.

A análise visual do cenário Inverso (Figura 4) expõe a vulnerabilidade dos algoritmos elementares diante de entradas desfavoráveis. Observa-se um crescimento vertiginoso nas curvas do Selection Sort (linha verde) e do Insertion Sort (linha laranja), que atingem picos superiores a 14.000ms e 11.000ms, respectivamente, para $N=10^5$. O comportamento do Insertion Sort é particularmente ilustrativo, pois materializa graficamente seu pior caso

teórico: a necessidade de deslocar cada novo elemento por toda a extensão do subvetor já ordenado. O Bubble Sort (linha azul), embora apresente desempenho ligeiramente superior aos outros dois quadráticos neste teste específico, acompanha a mesma tendência de degradação severa típica da complexidade O(n^2). Em contrapartida, QuickSort, MergeSort e HeapSort permanecem indistinguíveis próximos ao eixo horizontal, esse achatamento visual ocorre devido à discrepância de magnitude entre as complexidades: enquanto os métodos quadráticos explodem em tempo de execução, o crescimento log-linear (O(n log n)) é tão eficiente que, nesta escala gráfica ajustada para os piores casos, suas curvas parecem nulas. O destaque recai sobre o QuickSort, que manteve tempos de execução mínimos; isso evidencia que a estratégia de escolha do pivô foi eficaz em evitar o pior caso que a ordenação inversa causaria em uma implementação ingênua. Essa disparidade de escala ratifica a superioridade das abordagens de Divisão e Conquista (O(n log n)) e confirma a inadequação prática dos métodos simples para grandes volumes de dados inversamente ordenados.

4.2. CENÁRIO QUASE ORDENADO

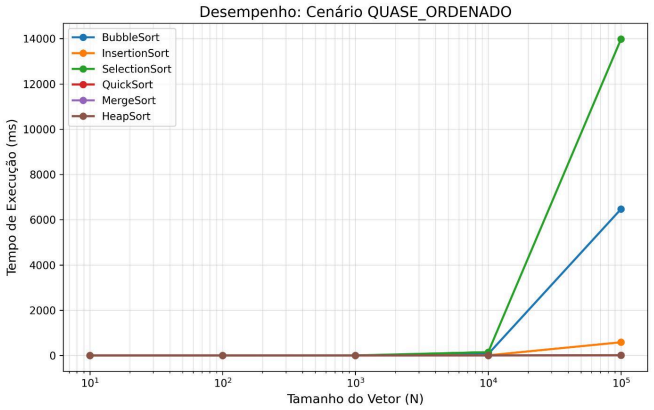

Figura 5 - Gráfico do Cenário Quase
Fonte: Adaptado pelo Autor. Gerado no Python.

A análise gráfica do cenário Quase Ordenado (Figura 5) ilustra a distinção crítica entre algoritmos adaptativos e rígidos. O Selection Sort (linha verde) isola-se negativamente como o pior desempenho, atingindo ~14.000ms para N=10^5; sua curva ascendente confirma a insensibilidade do algoritmo à pré-ordenação, visto que ele executa a totalidade de comparações (O(n^2)) independentemente do estado dos dados. O Bubble Sort (linha azul) apresenta uma melhora relativa quando comparado aos cenários anteriores, pois realiza

menos trocas, mas ainda sofre com o gargalo estrutural de comparações excessivas. O comportamento mais notável, contudo, é a "redenção" do Insertion Sort (linha laranja). Diferente do cenário Inverso, aqui sua curva colapsa para próximo do eixo horizontal, juntando-se ao grupo de elite (QuickSort, MergeSort, HeapSort). Isso ocorre porque, em vetores quase ordenados, o número de inversões é baixo, permitindo que o Insertion Sort opere próximo de sua complexidade linear (O(n)). O gráfico, portanto, valida visualmente a teoria de que algoritmos simples podem superar ou igualar métodos sofisticados (O(n log n)) quando aplicados em contextos de refinamento de dados onde a entropia é baixa.

4.3. CENÁRIO ALEATÓRIO

No cenário aleatório (Figura 6), o comportamento dos algoritmos foi mais equilibrado e próximo do esperado teoricamente. QuickSort, mais uma vez, teve excelente desempenho e se manteve entre os mais rápidos em quase todos os tamanhos, mostrando sua eficiência média. MergeSort e HeapSort também apresentaram resultados sólidos e estáveis, reafirmando sua complexidade O(n log n) sem grande variação. Já os algoritmos quadráticos — Bubble Sort, Insertion Sort e Selection Sort — tiveram aumento expressivo de tempo conforme o tamanho cresceu, tornando-se rapidamente impraticáveis para vetores grandes. O Insertion Sort, embora quadrático, ainda performou relativamente melhor entre os $O(n^2)$, confirmando ser o mais eficiente desse grupo quando os dados não estão totalmente desordenados. No geral, o cenário aleatório reforça a robustez dos algoritmos mais avançados e evidencia o custo dos métodos simples em escalas maiores.

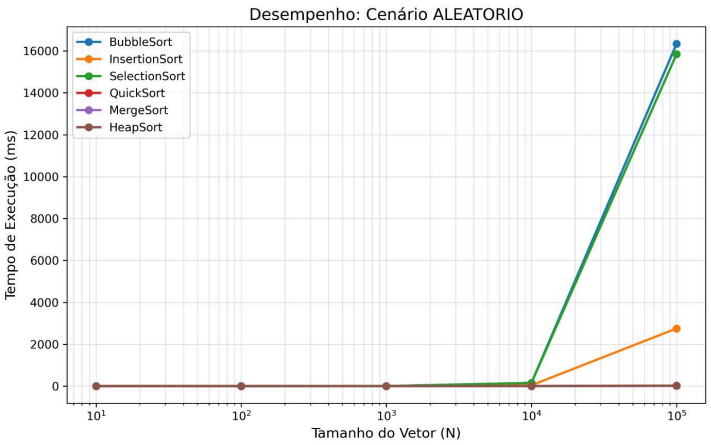

Figura 6 - Gráfico do Cenário Aleatório
Fonte: Adaptado pelo Autor. Gerado no Python.

4.4. LISTAGEM TABULAR (DADOS BRUTOS EXTRAÍDOS)

A seção a seguir apresenta a listagem tabular dos dados brutos obtidos durante a execução dos experimentos. Esses valores são exibidos sem qualquer tratamento ou interpretação adicional, servindo como base primária para todas as análises, gráficos e conclusões desenvolvidas ao longo do trabalho.

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
BubbleSort	10	ORDENADO	8800	0	88	45
InsertionSort	10	ORDENADO	11500	0	115	9
SelectionSort	10	ORDENADO	11900	0	119	45
QuickSort	10	ORDENADO	30800	0	308	20
MergeSort	10	ORDENADO	30200	0	302	19
HeapSort	10	ORDENADO	37300	0	373	41
BubbleSort	100	ORDENADO	527400	0	5274	4950
InsertionSort	100	ORDENADO	39700	0	397	99
SelectionSort	100	ORDENADO	278600	0	2786	4950
QuickSort	100	ORDENADO	194400	0	1944	703
MergeSort	100	ORDENADO	142000	0	1420	356
HeapSort	100	ORDENADO	368500	0	3685	1081
BubbleSort	1000	ORDENADO	7392500	7	3925	499500
InsertionSort	1000	ORDENADO	133400	0	1334	999
SelectionSort	1000	ORDENADO	10089900	10	899	499500
QuickSort	1000	ORDENADO	1433200	1	4332	11604
MergeSort	1000	ORDENADO	1250700	1	2507	5044
HeapSort	1000	ORDENADO	542700	0	5427	17583
BubbleSort	10000	ORDENADO	24919200	24	9192	49995000
InsertionSort	10000	ORDENADO	1395300	1	3953	9999
SelectionSort	10000	ORDENADO	29513700	29	5137	49995000

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
QuickSort	10000	ORDENADO	7403700	7	4037	161069
MergeSort	10000	ORDENADO	2939300	2	9393	69008
HeapSort	10000	ORDENADO	7481900	7	4819	244460
BubbleSort	100000	ORDENADO	2738986800	2738	9868	4999950000
InsertionSort	100000	ORDENADO	8625400	8	6254	99999
SelectionSort	100000	ORDENADO	2564629700	2564	6297	4999950000
QuickSort	100000	ORDENADO	5953300	5	9533	2114356
MergeSort	100000	ORDENADO	13734000	13	7340	853904
HeapSort	100000	ORDENADO	15758800	15	7588	3112517
BubbleSort	10	INVERSO	104500	0	1045	45
InsertionSort	10	INVERSO	22600	0	226	45
SelectionSort	10	INVERSO	57200	0	572	45
QuickSort	10	INVERSO	2100	0	21	22
MergeSort	10	INVERSO	74000	0	740	15
HeapSort	10	INVERSO	33800	0	338	35
BubbleSort	100	INVERSO	975900	0	9759	4950
InsertionSort	100	INVERSO	55500	0	555	4950
SelectionSort	100	INVERSO	224600	0	2246	4950
QuickSort	100	INVERSO	6900	0	69	681
MergeSort	100	INVERSO	130100	0	1301	316
HeapSort	100	INVERSO	8200	0	82	944

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
BubbleSort	1000	INVERSO	19607700	19	6077	499500
InsertionSort	1000	INVERSO	6736000	6	7360	499500
SelectionSort	1000	INVERSO	7234800	7	2348	499500
QuickSort	1000	INVERSO	58700	0	587	11611
MergeSort	1000	INVERSO	1591400	1	5914	4932
HeapSort	1000	INVERSO	92500	0	925	15965
BubbleSort	10000	INVERSO	86325700	86	3257	49995000
InsertionSort	10000	INVERSO	125095500	125	955	49995000
SelectionSort	10000	INVERSO	157658500	157	6585	49995000
QuickSort	10000	INVERSO	604200	0	6042	155285
MergeSort	10000	INVERSO	17312700	17	3127	64608
HeapSort	10000	INVERSO	1122600	1	1226	226682
BubbleSort	100000	INVERSO	9217462500	9217	4625	4999950000
InsertionSort	100000	INVERSO	11710745100	11710	7451	4999950000
SelectionSort	100000	INVERSO	14162930900	14162	9309	4999950000
QuickSort	100000	INVERSO	7320000	7	3200	1936294
MergeSort	100000	INVERSO	14871300	14	8713	815024
HeapSort	100000	INVERSO	19690300	19	6903	2926640
BubbleSort	10	ALEATORIO	1500	0	15	45
InsertionSort	10	ALEATORIO	3000	0	30	29
SelectionSort	10	ALEATORIO	1200	0	12	45

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
QuickSort	10	ALEATORIO	2100	0	21	30
MergeSort	10	ALEATORIO	3500	0	35	24
HeapSort	10	ALEATORIO	2200	0	22	38
BubbleSort	100	ALEATORIO	96400	0	964	4950
InsertionSort	100	ALEATORIO	4400	0	44	2688
SelectionSort	100	ALEATORIO	19500	0	195	4950
QuickSort	100	ALEATORIO	8000	0	80	599
MergeSort	100	ALEATORIO	12400	0	124	548
HeapSort	100	ALEATORIO	9400	0	94	1034
BubbleSort	1000	ALEATORIO	1408600	1	4086	499500
InsertionSort	1000	ALEATORIO	264300	0	2643	251948
SelectionSort	1000	ALEATORIO	1342700	1	3427	499500
QuickSort	1000	ALEATORIO	80800	0	808	10747
MergeSort	1000	ALEATORIO	117900	0	1179	8670
HeapSort	1000	ALEATORIO	107700	0	1077	16834
BubbleSort	10000	ALEATORIO	155028800	155	288	49995000
InsertionSort	10000	ALEATORIO	38308300	38	3083	25293137
SelectionSort	10000	ALEATORIO	148377400	148	3774	49995000
QuickSort	10000	ALEATORIO	858700	0	8587	152418
MergeSort	10000	ALEATORIO	1309600	1	3096	120521
HeapSort	10000	ALEATORIO	1415100	1	4151	235279

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
BubbleSort	100000	ALEATORIO	16340744300	16340	7443	4999950000
InsertionSort	100000	ALEATORIO	2752645200	2752	6452	2502843024
SelectionSort	100000	ALEATORIO	15847665900	15847	6659	4999950000
QuickSort	100000	ALEATORIO	10512000	10	5120	1985135
MergeSort	100000	ALEATORIO	17082400	17	824	1536376
HeapSort	100000	ALEATORIO	25426500	25	4265	3019431
BubbleSort	10	QUASE_ORDENADO	1600	0	16	45
InsertionSort	10	QUASE_ORDENADO	800	0	8	24
SelectionSort	10	QUASE_ORDENADO	1000	0	10	45
QuickSort	10	QUASE_ORDENADO	2400	0	24	24
MergeSort	10	QUASE_ORDENADO	3000	0	30	25
HeapSort	10	QUASE_ORDENADO	48600	0	486	40
BubbleSort	100	QUASE_ORDENADO	8500	0	85	4950
InsertionSort	100	QUASE_ORDENADO	1600	0	16	519
SelectionSort	100	QUASE_ORDENADO	13600	0	136	4950
QuickSort	100	QUASE_ORDENADO	6400	0	64	639
MergeSort	100	QUASE_ORDENADO	9400	0	94	455
HeapSort	100	QUASE_ORDENADO	54900	0	549	1082
BubbleSort	1000	QUASE_ORDENADO	532100	0	5321	499500
InsertionSort	1000	QUASE_ORDENADO	62800	0	628	57177
SelectionSort	1000	QUASE_ORDENADO	1363400	1	3634	499500

Algoritmo	Tamanho	Cenario	Tempo(ns)	Tempo(ms)	Comparacoes	Atribuicoes
QuickSort	1000	QUASE_ORDENADO	60600	0	606	10007
MergeSort	1000	QUASE_ORDENADO	195900	0	1959	8052
HeapSort	1000	QUASE_ORDENADO	296300	0	2963	17386
BubbleSort	10000	QUASE_ORDENADO	65879500	65	8795	49995000
InsertionSort	10000	QUASE_ORDENADO	6427000	6	4270	5784049
SelectionSort	10000	QUASE_ORDENADO	149481400	149	4814	49995000
QuickSort	10000	QUASE_ORDENADO	702700	0	7027	166660
MergeSort	10000	QUASE_ORDENADO	1141700	1	1417	115844
HeapSort	10000	QUASE_ORDENADO	1011200	1	112	242540
BubbleSort	100000	QUASE_ORDENADO	6470711700	6470	7117	4999950000
InsertionSort	100000	QUASE_ORDENADO	583193600	583	1936	576995345
SelectionSort	100000	QUASE_ORDENADO	13981284000	13981	2840	4999950000
QuickSort	100000	QUASE_ORDENADO	7518500	7	5185	1927393
MergeSort	100000	QUASE_ORDENADO	11926600	11	9266	1481622
HeapSort	100000	QUASE_ORDENADO	11999400	11	9994	3095331

5. CONCLUSÃO

O desenvolvimento deste trabalho permitiu analisar de forma prática o comportamento dos principais algoritmos de ordenação em diferentes cenários de entrada. A comparação dos

resultados evidenciou como a organização inicial dos dados influencia diretamente o desempenho, destacando a superioridade dos algoritmos O(n log n), como QuickSort e MergeSort, especialmente em conjuntos grandes. Também foi possível observar as limitações dos métodos quadráticos e identificar situações em que algoritmos simples, como o Insertion Sort, podem se destacar. Entre as principais dificuldades encontradas estiveram a implementação consistente das medições de tempo, garantindo resultados confiáveis, e o tratamento adequado dos vetores gerados para cada cenário, de forma a manter a fidelidade dos testes. No geral, o trabalho proporcionou uma compreensão aprofundada da eficiência prática dos algoritmos e reforçou a importância da análise experimental na validação de conceitos teóricos.

6. REFERÊNCIAS

GEKSFORGEEKS. *Sorting Algorithms*. Disponível em: https://www.geeksforgeeks.org/sorting-algorithms/. Acesso em: 24 nov. 2025.

W3SCHOOLS. *W3Schools — Online Web Tutorials*. Disponível em: https://www.w3schools.com/. Acesso em: 24 nov. 2025.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algorithms: Theory and Practice (Introduction to Algorithms)*. Cambridge: MIT Press, [ano de publicação, se disponível].