

Matemática Discreta

Maria Eduarda Moraes

17/10/2023

PARTE 1

Código fonte:

Class MergeSort ->

```
public class MergeSort {

    private static long comparisons = 0; // Variável para contar as comparações
    private static long startTime; // Variável para medir o tempo de início
    private static long executionTime;

    public static void mergeSort(int[] arr) {
        startTime = System.nanoTime(); // Inicia a medição de tempo
        mergeSortRecursive(arr);
        long endTime = System.nanoTime(); // Finaliza a medição de tempo
        executionTime = endTime - startTime;
        /*System.out.println("Mergesort:");
        System.out.println("Número de comparações: " + comparisons);
        System.out.println("Tempo gasto (em nanossegundos): " + executionTime + "\n");*/
    }

    private static void mergeSortRecursive(int[] arr) {
        if (arr.length > 1) {
            int mid = arr.length / 2;
            int[] left = new int[mid];
            int[] right = new int[arr.length - mid];

            // Copia elementos para os subarrays "left" e "right"
            System.arraycopy(src:arr, srcPos:0, dest:left, destPos:0, length:mid);
            for (int i = mid; i < arr.length; i++) {
                right[i - mid] = arr[i];
            }

            // Chama recursivamente o Mergesort para os subarrays
            mergeSortRecursive(arr:left);
            mergeSortRecursive(arr:right);

            int i = 0, j = 0, k = 0;
            while (i < left.length && j < right.length) {
                if (left[i] < right[j]) {
                    arr[k] = left[i];
                    i++;
                } else {
                    arr[k] = right[j];
                    j++;
                }
                k++;
                comparisons++; // Incrementa as comparações aqui
            }
        }
    }
}
```

```

        // Copia os elementos restantes, se houver, de "left" e "right" para "arr"
        while (i < left.length) {
            arr[k] = left[i];
            i++;
            k++;
        }
        while (j < right.length) {
            arr[k] = right[j];
            j++;
            k++;
        }
    }
}

public static long getComparisons() {
    return comparisons;
}

public static long getExecutionTime() {
    return executionTime;
}
}

```



Class QuickSort ->

```

public class QuickSort {

    private static long comparisons = 0; // Variável para contar as comparações
    private static long startTime; // Variável para medir o tempo de início
    private static long executionTime; // Variável para armazenar o tempo gasto

    public static void quickSort(int[] arr, int low, int high) {
        comparisons = 0; // Zera a contagem de comparações
        startTime = System.nanoTime(); // Inicia a medição de tempo
        quickSortHelper(arr, low, high);
        long endTime = System.nanoTime(); // Finaliza a medição de tempo
        executionTime = endTime - startTime;
    }

    private static void quickSortHelper(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSortHelper(arr, low, pi - 1);
            quickSortHelper(arr, pi + 1, high);
        }
    }
}

```

```

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            comparisons++; // Incrementa a contagem de comparações
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    comparisons++; // Incrementa a contagem de comparações
    return i + 1;
}

public static long getComparisons() {
    return comparisons;
}

public static long getExecutionTime() {
    return executionTime;
}
}

```



Class InsertionSortRecursive ->

```

public class InsertionSortRecursive {

    private static long comparisons = 0; // Variável para contar as comparações
    private static long startTime; // Variável para medir o tempo de início
    private static long executionTime; // Variável para armazenar o tempo gasto

    public static void insertionSortRecursive(int[] arr, int n) {
        comparisons = 0; // Zera a contagem de comparações
        startTime = System.nanoTime(); // Inicia a medição de tempo
        insertionSortRecursiveHelper(arr, n);
        long endTime = System.nanoTime(); // Finaliza a medição de tempo
        executionTime = endTime - startTime;
    }
}

```

```

private static void insertionSortRecursiveHelper(int[] arr, int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
            comparisons++; // Incrementa a contagem de comparações
        }

        arr[j + 1] = key;
    }
}

public static long getComparisons() {
    return comparisons;
}

public static long getExecutionTime() {
    return executionTime;
}
}

```



Class Main ->

```

public class MatematicaDiscretaTrab {

    public static void main(String[] args) {
        int[] inputSizes = {10, 50, 100, 1000, 10000, 100000};

        createAndShowChart(algorithm: "Mergesort", inputSizes);
        createAndShowChart(algorithm: "Inserção Recursivo", inputSizes);
        createAndShowChart(algorithm: "Quicksort", inputSizes);

        printTable(algorithm: "Mergesort", sizes: inputSizes);
        printTable(algorithm: "Inserção Recursivo", sizes: inputSizes);
        printTable(algorithm: "Quicksort", sizes: inputSizes);
    }
}

```

```

public static void createAndShowChart(String algorithm, int[] inputSizes) {
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();

    for (int i = 0; i < inputSizes.length; i++) {
        int size = inputSizes[i];

        double expectedComparisons = calculateExpectedComparisons(algorithm, size);
        double actualComparisons = runSortingAlgorithm(algorithm, size);

        dataset.addValue(value: expectedComparisons, algorithm + " Esperado", columnKey: String.valueOf(i: size));
        dataset.addValue(value: actualComparisons, algorithm + " Alcançado", columnKey: String.valueOf(i: size));
    }

    JFreeChart chart = ChartFactory.createLineChart(
        "Comparação de Comparações Esperadas e Alcançadas - " + algorithm,
        categoryAxisLabel: "Tamanho de Entrada",
        valueAxisLabel: "Comparações",
        dataset,
        orientation: PlotOrientation.VERTICAL,
        legend: true,
        tooltips: true,
        urls: false
    );
}

```

```

CategoryPlot plot = (CategoryPlot) chart.getPlot();
CategoryAxis categoryAxis = plot.getDomainAxis();
categoryAxis.setCategoryLabelPositions(positions: CategoryLabelPositions.STANDARD);

ChartPanel chartPanel = new ChartPanel(chart);
chartPanel.setPreferredSize(new java.awt.Dimension(width: 800, height: 600));
ApplicationFrame frame = new ApplicationFrame("Comparação de Comparações - " + algorithm);
frame.setContentPane(contentPane: chartPanel);
frame.pack();
RefineryUtilities.centerFrameOnScreen(frame);
frame.setVisible(b: true);
}

```

```

public static double calculateExpectedComparisons(String algorithm, int size) {
    double constant = 2.5;
    if (null != algorithm) {
        switch (algorithm) {
            case "Mergesort" -> {
                return constant * size * Math.log(a: size) / Math.log(a: 2);
            }
            case "Inserção Recursivo" -> {
                return constant * size;
            }
            case "Quicksort" -> {
                return constant * size * Math.log(a: size) / Math.log(a: 2);
            }
            default -> {
            }
        }
    }
    return 0;
}

```

```
public static void printTable(String algorithm, int[] sizes) {
    System.out.println("Resultados para " + algorithm + "\n");
    System.out.println("Tamanho da Entrada\tComparações Esperadas\tComparações Alcançadas\tTempo (ms)");

    for (int size : sizes) {
        int[] randomArray = generateRandomArray(size);
        int[] array = Arrays.copyOf(original, randomArray.length);
        runSortingAlgorithm(array, algorithm);

        double expectedComparisons = calculateExpectedComparisons(algorithm, size);
        double actualComparisons = getComparisonsForAlgorithm(algorithm);
        long executionTime = getExecutionTimeForAlgorithm(algorithm);

        System.out.printf(format: "%d\t\t\t%.0f\t\t\t%.0f\t\t\t%d\n", args: size,
                           args: expectedComparisons, args: actualComparisons, args: executionTime);
    }
    System.out.println();
}
```

```
public static double runSortingAlgorithm(String algorithm, int size) {
    int[] randomArray = generateRandomArray(size);
    int[] array = Arrays.copyOf(original, randomArray.length);
    runSortingAlgorithm(array, algorithm);

    return getComparisonsForAlgorithm(algorithm);
}

public static void runSortingAlgorithm(int[] array, String algorithm) {
    switch (algorithm) {
        case "Mergesort" ->
            MergeSort.mergeSort(arr:array);
        case "Inserção Recursivo" ->
            InsertionSortRecursive.insertionSortRecursive(arr:array, n: array.length);
        case "Quicksort" ->
            QuickSort.quickSort(arr:array, low:0, array.length - 1);
        default -> {
        }
    }
}
```

```
public static double getComparisonsForAlgorithm(String algorithm) {
    switch (algorithm) {
        case "Mergesort" -> {
            return MergeSort.getComparisons();
        }
        case "Inserção Recursivo" -> {
            return InsertionSortRecursive.getComparisons();
        }
        case "Quicksort" -> {
            return QuickSort.getComparisons();
        }
        default -> {
        }
    }
    return 0;
}
```

```
public static long getExecutionTimeForAlgorithm(String algorithm) {  
    switch (algorithm) {  
        case "Mergesort" -> {  
            return MergeSort.getExecutionTime();  
        }  
        case "Inserção Recursivo" -> {  
            return InsertionSortRecursive.getExecutionTime();  
        }  
        case "Quicksort" -> {  
            return QuickSort.getExecutionTime();  
        }  
        default -> {  
        }  
    }  
    return 0;  
}  
  
public static int[] generateRandomArray(int size) {  
    int[] arr = new int[size];  
    Random random = new Random();  
    for (int i = 0; i < size; i++) {  
        arr[i] = random.nextInt(bound: 1000);  
    }  
    return arr;  
}  
}
```

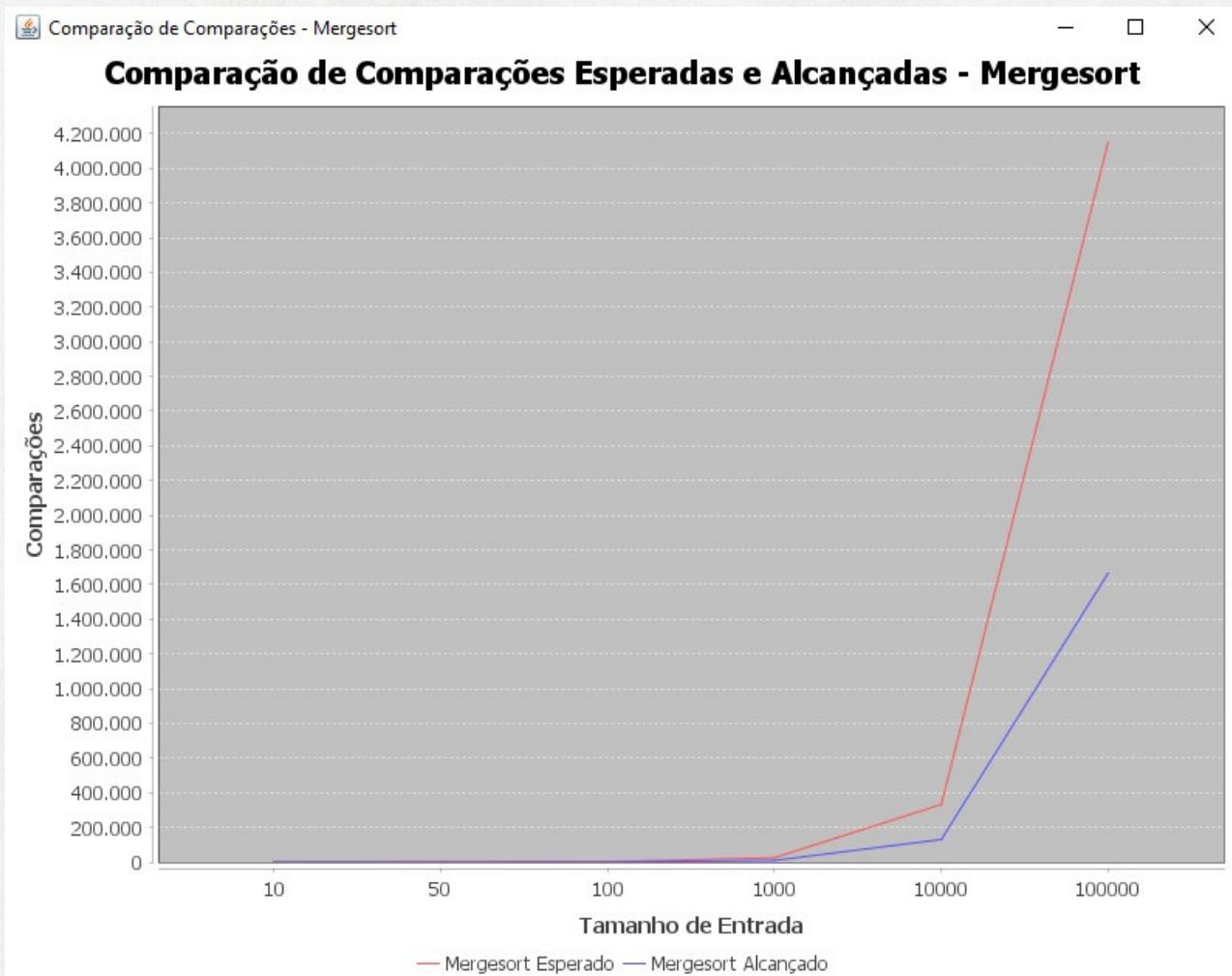
Código fonte completo no GitHub:



ANALISANDO RESULTADOS

MergeSort:

Resultados para Mergesort			
Tamanho da Entrada	Comparações Esperadas	Comparações Alcançadas	Tempo (ms)
10	83	1666412	13399
50	705	1666630	3600
100	1661	1667179	6000
1000	24914	1675875	62500
10000	332193	1796290	732700
100000	4152410	3332476	10132000



QuickSort:

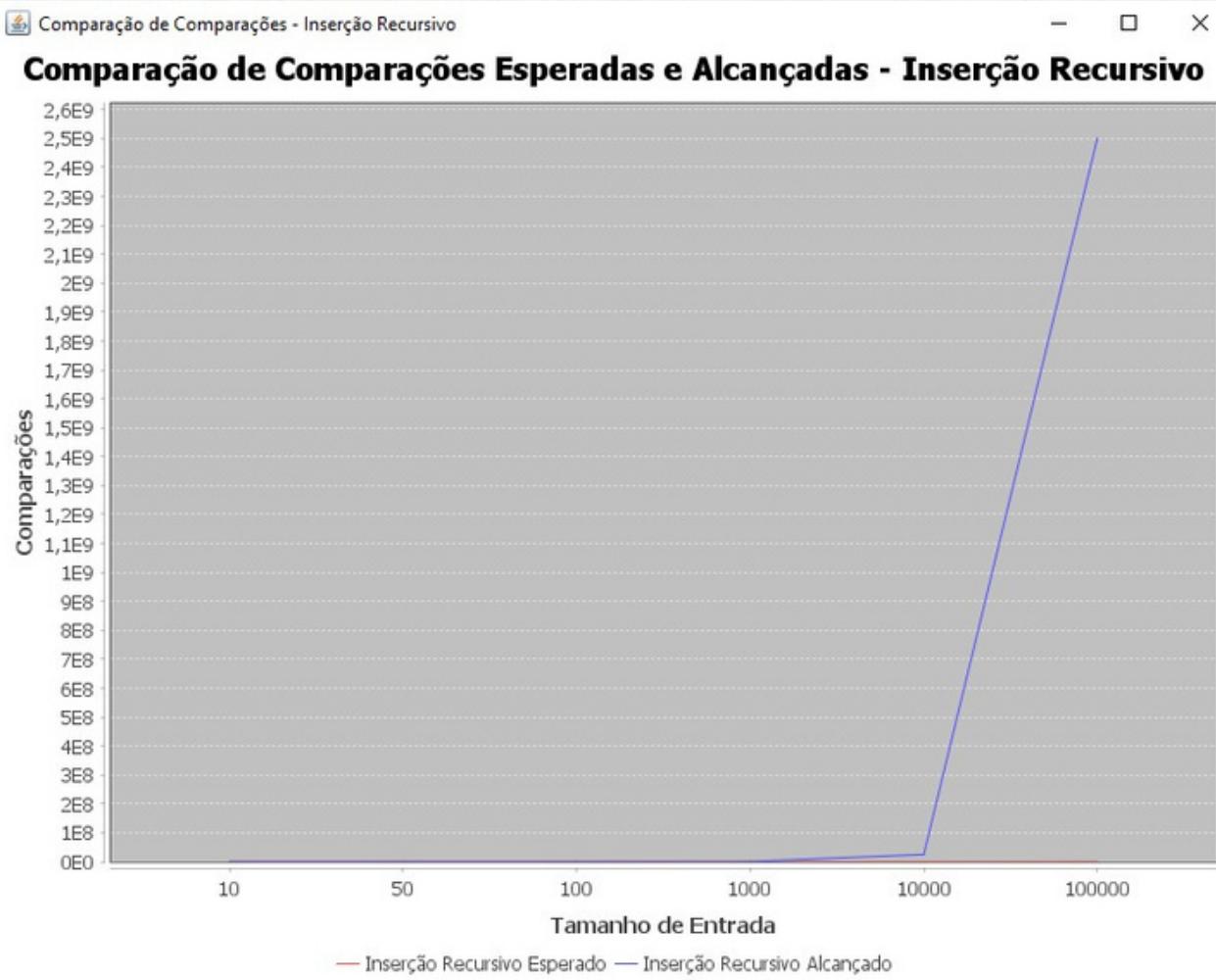
Resultados para Quicksort			
Tamanho da Entrada	Comparações Esperadas	Comparações Alcançadas	Tempo (ms)
10	83	19	6500
50	705	187	4000
100	1661	351	3100
1000	24914	6575	39701
10000	332193	67878	393800
100000	4152410	779575	5172999



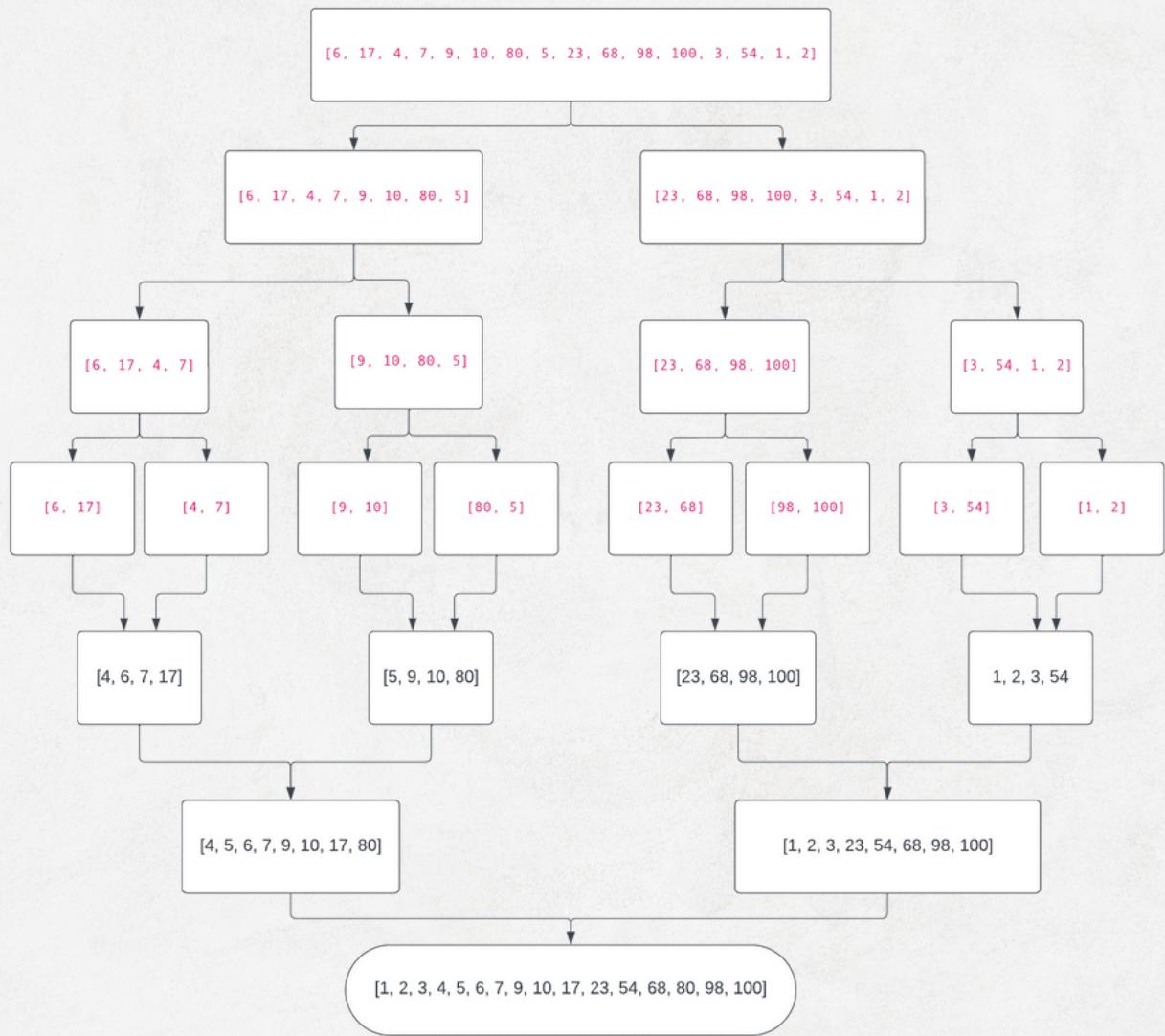
InsertionSortRecursive:

Resultados para Inserção Recursivo

Tamanho da Entrada	Comparações Esperadas	Comparações Alcançadas	Tempo (ms)
10	25	24	400
50	125	621	900
100	250	2213	1701
1000	2500	252187	132300
10000	25000	24943664	11464101
100000	250000	2488827409	1149057100



PARTE 2



PARTE 2

Cálculo do Número de Comparações no Mergesort

O Mergesort é um algoritmo de ordenação que divide uma lista em duas metades, ordena cada metade e, em seguida, mescla as duas metades para produzir a lista final ordenada. Um aspecto fundamental na análise do desempenho do Mergesort é o número de comparações (ou trocas) que ocorrem durante a ordenação.

A relação de recorrência do Mergesort é dada por: $T(n) = 2 * T(n/2) + n$

Onde:

- $T(n)$ é o número de comparações (ou trocas) necessárias para ordenar uma lista de tamanho n .
- A primeira parte ($2 * T(n/2)$) representa o número de comparações necessárias para ordenar as duas metades da lista.
- A segunda parte (n) representa o número de comparações necessárias para mesclar as duas metades ordenadas.

A solução dessa relação de recorrência é $O(n * \log_2(n))$, onde "O" denota a notação assintótica de ordem grande. Portanto, o número de comparações esperado no Mergesort é proporcional a $n * \log_2(n)$.

Exemplo Prático

Para ilustrar o cálculo do número de comparações no Mergesort, consideremos a ordenação de uma lista de 16 elementos:

6, 17, 4, 6, 9, 10, 80, 5, 23, 68, 98, 100, 3, 54, 1, 2

1. **Estado Inicial:** A lista inicial possui 16 elementos, portanto, não há trocas neste estágio.
2. **Divisão em Pares e Mesclagem (etapa 1 a 4):** O número de comparações nas etapas de divisão e mesclagem é dado pelo número de pares de elementos (2 elementos cada) que são comparados. A primeira etapa envolve 8 comparações (4 pares de elementos), a segunda etapa envolve 12 comparações (6 pares de elementos), a terceira etapa envolve 14 comparações (7 pares de elementos) e a quarta etapa envolve 15 comparações (8 pares de elementos).
3. **Mesclagem Final (etapa 5):** Esta etapa envolve 14 comparações (7 pares de elementos).

CONCLUSÃO

Somando todas as comparações das etapas, o número total de comparações (ou trocas) é $8 + 12 + 14 + 15 + 14 = 63$ comparações. Portanto, para a lista de 16 elementos fornecida, o número exato de trocas (comparações) feitas pelo Mergesort é 63.

Isso está em linha com a análise teórica, que prevê um número de comparações próximo a **$16 * \log_2(16) = 64$** .

O cálculo do número de comparações é uma maneira essencial de avaliar o desempenho do Mergesort e entender sua eficiência em ordenar listas de diferentes tamanhos.

Neste estudo individual, explorei a análise e a visualização do desempenho de algoritmos de ordenação, com foco no Mergesort, Inserção Recursivo e Quicksort. Utilizei uma abordagem quantitativa para comparar esses algoritmos em termos de comparações esperadas e alcançadas, além do tempo de execução. Foi evidenciado que a análise teórica dos algoritmos de ordenação fornece uma estimativa sólida do desempenho esperado. O Mergesort, por exemplo, demonstrou uma complexidade assintótica de $O(n * \log_2(n))$, o que está alinhado com as comparações observadas no experimento.

Um aspecto importante deste estudo foi a utilização da biblioteca JFreeChart para criar gráficos que representam visualmente a relação entre o tamanho da entrada e o número de comparações esperadas e alcançadas. Os gráficos facilitaram a compreensão das diferenças no desempenho entre os algoritmos e destacaram como as comparações esperadas se relacionam com as comparações alcançadas em vários tamanhos de entrada.

Também conduzi uma análise prática do Mergesort, onde ordenei uma lista de 16 números e contei o número de trocas realizadas. Este resultado prático foi comparado com a ordem de complexidade calculada pela relação de recorrência do Mergesort, confirmou a validade da análise teórica.

No geral, esta pesquisa individual demonstra a importância da análise e visualização do desempenho de algoritmos de ordenação para avaliar eficiência e tomar decisões informadas ao escolher algoritmos para aplicações do mundo real. Além disso, a integração da biblioteca JFreeChart permitiu a representação visual dos resultados, tornando a comunicação das conclusões mais eficaz.