ECM306 – TÓPICOS AVANÇADOS EM ESTRUTURA DE DADOS ENGENHARIA DA COMPUTAÇÃO – 3ª SÉRIE – 2025 – Prof. Calvetti

Exercícios propostos da Aula 08

Pedro Wilian Palumbo Bevilacqua – RA: 23.01307-9

Exercício 1 – Merge Sort

```
public class MergeSort {

    public void mergeSort(int[] arr, int start, int end) {
        if (start < end) {
            int middle = (start + end) / 2;

            mergeSort(arr, start, middle);
            mergeSort(arr, middle + 1, end);

            merge(arr, start, middle, end);
        }
    }

    public void merge(int[] arr, int start, int middle, int end) {
        int[] helper = new int[arr.length];

        if (end + 1 - start >= 0) System.arraycopy(arr, start, helper,
start, end + 1 - start);

        int helperLeft = start;
        int helperRight = middle + 1;
        int current = start;

        while (helperLeft <= middle && helperRight <= end) {
            if (helper[helperLeft] <= helper[helperRight]) {
                arr[current] = helper[helperLeft];
                helperLeft++;
            } else {
                arr[current] = helper[helperRight];
                helperRight++;
            }
            current++;
        }

        int remaining = middle - helperLeft;
        if (remaining + 1 >= 0) System.arraycopy(helper, helperLeft, arr,
current, remaining + 1);
    }

    public static void main(String[] args) {
```

```
        int[] arr = {38, 27, 43, 3, 9, 82, 10};

        System.out.println("Array original:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();

        MergeSort sorter = new MergeSort();
        sorter.mergeSort(arr, 0, arr.length - 1);

        System.out.println("Array ordenado:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

A ordem de complexidade do algoritmo Merge Sort, considerando o pior caso, é **O(n log n)**. Isso ocorre porque o algoritmo segue a estratégia de dividir para conquistar (divide and conquer), em que o vetor é recursivamente dividido em duas metades até que cada subvetor contenha apenas um elemento. Esse processo de divisão acontece em $\log_2(n)$ níveis. Em cada nível da recursão, o algoritmo realiza um processo de mesclagem (merge) dos subvetores, que exige tempo linear (O(n)) para combinar todos os elementos. Como o processo de mesclagem ocorre em todos os níveis da divisão, o custo total no pior caso é a multiplicação do número de níveis (log n) pelo custo de cada nível (n), resultando em uma complexidade de O(n log n). Essa complexidade é garantida independentemente da ordem inicial dos elementos, sendo uma das principais vantagens do Merge Sort em comparação com algoritmos como o Insertion Sort ou o Bubble Sort, que apresentam desempenho quadrático no pior caso.

Exercício 2 – Busca Binária Recursiva

```
public class BuscaBinariaRecursiva {

    static int[] A = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}; // Array
ordenado
    static int indice = -1;
    static int nComparacoes = 0;

    public static void binSearch(int item, int begin, int end) {
        int metade = (begin + end) / 2;
```

```
        if (begin > end) { // caso base
            indice = -1;
            nComparacoes++;
            return;
        }

        if (A[metade] == item) {
            indice = metade;
            nComparacoes++;
            return;
        }

        if (A[metade] < item) {
            nComparacoes++;
            binSearch(item, metade + 1, end);
        } else {
            nComparacoes++;
            binSearch(item, begin, metade);
        }
    }

    public static void main(String[] args) {
        int itemProcurado = 14;

        binSearch(itemProcurado, 0, A.length - 1);

        if (indice != -1) {
            System.out.println("Item encontrado no índice: " + indice);
        } else {
            System.out.println("Item não encontrado.");
        }

        System.out.println("Número de comparações: " + nComparacoes);
    }
}
```

A ordem de complexidade do algoritmo de busca binária recursiva é **O(log n)**, pois a cada chamada recursiva o espaço de busca é dividido pela metade. Isso significa que, partindo de um vetor com n elementos, o algoritmo realiza no máximo $\log_2(n)$ comparações até encontrar o elemento desejado ou determinar que ele não está presente. Essa eficiência se dá porque a busca binária elimina metade dos elementos possíveis a cada passo, reduzindo drasticamente o número de operações em comparação com uma busca linear. No entanto, essa complexidade só se mantém válida quando o vetor está previamente ordenado, condição essencial para o funcionamento correto do algoritmo.