Exercícios Propostos - Aula 08

Felipe Fazio da Costa

RA: 23.00055-4
Disciplina: ECM306 - Tópicos Avançados em Estrutura de Dados

Implementação em Java – Merge Sort

```java
public class MergeSort {

    public static void mergeSort(int[] array, int esquerda, int direita) {
        if (esquerda < direita) {
            int meio = (esquerda + direita) / 2;
            mergeSort(array, esquerda, meio);
            mergeSort(array, meio + 1, direita);
            merge(array, esquerda, meio, direita);
        }
    }

    private static void merge(int[] array, int esquerda, int meio, int direita) {
        int n1 = meio - esquerda + 1;
        int n2 = direita - meio;

        int[] esquerdaArray = new int[n1];
        int[] direitaArray = new int[n2];

        for (int i = 0; i < n1; i++)
            esquerdaArray[i] = array[esquerda + i];
        for (int j = 0; j < n2; j++)
            direitaArray[j] = array[meio + 1 + j];

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

        while (i < n1 && j < n2) {
            if (esquerdaArray[i] <= direitaArray[j]) {
                array[k] = esquerdaArray[i];
                i++;
            } else {
                array[k] = direitaArray[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            array[k] = esquerdaArray[i];
            i++;
            k++;
        }
```

```
        while (j < n2) {
            array[k] = direitaArray[j];
            j++;
            k++;
        }
    }

    public static void main(String[] args) {
        int[] array = {38, 27, 43, 3, 9, 82, 10};
        mergeSort(array, 0, array.length - 1);

        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

---

2. Definição da Recorrência

Seja `T(n)` o tempo de execução do Merge Sort para um vetor de tamanho `n`. O algoritmo:

- Divide o vetor em dois subvetores de tamanho `n/2` → 2 chamadas recursivas: `2T(n/2)`
- Mescla os dois subvetores em tempo linear → `O(n)`

Logo, a recorrência é:

$$ T(n) = 2T\left(\frac{n}{2}\right) + cn $$

onde `c` é uma constante positiva que representa o custo da mesclagem.

---

2. Resolução da Recorrência – Método da Árvore de Recursão

Vamos expandir a recorrência para visualizar os custos por nível da árvore:

Nível 0 (raiz):

$$ T(n) = 2T(n/2) + cn $$

Nível 1:

$$ 2T(n/2) = 4T(n/4) + 2c(n/2) = 4T(n/4) + cn $$

Nível 2:

$$ 4T(n/4) = 8T(n/8) + 4c(n/4) = 8T(n/8) + cn $$

...

Nível `k`:

$$ T(n) = 2^k T\left(\frac{n}{2^k}\right) + k \cdot cn $$

A recursão termina quando `n/2^k = 1` → ou seja, `k = log₂(n)`

Substituindo:

$$ T(n) = nT(1) + cn \log_2(n) $$

Como `T(1)` é constante (`O(1)`), temos:

$$ T(n) = O(n \log n) $$

---

3. Conclusão

A dedução formal da recorrência confirma que a **complexidade de tempo** do Merge Sort no pior caso é:

$$ \boxed{O(n \log n)} $$

Segue a implementação **recursiva** do algoritmo de **Busca Binária** em Java, seguida da análise da **ordem de complexidade** com justificativa formal.

Implementação em Java – Busca Binária Recursiva

```java
public class BuscaBinaria {

    public static int buscaBinariaRecursiva(int[] array, int valor, int esquerda, int direita) {
        if (esquerda > direita) {
            return -1; // Valor não encontrado
        }

        int meio = (esquerda + direita) / 2;

        if (array[meio] == valor) {
            return meio;
        } else if (valor < array[meio]) {
            return buscaBinariaRecursiva(array, valor, esquerda, meio - 1);
        } else {
            return buscaBinariaRecursiva(array, valor, meio + 1, direita);
        }
    }

    public static void main(String[] args) {
        int[] array = {2, 4, 6, 8, 10, 12, 14, 16};
        int valor = 10;
        int resultado = buscaBinariaRecursiva(array, valor, 0, array.length - 1);

        if (resultado != -1) {
            System.out.println("Valor encontrado na posição: " + resultado);
        } else {
            System.out.println("Valor não encontrado.");
```

```
            }
        }
    }
```

Ordem de Complexidade

Tempo – Pior Caso

A cada chamada recursiva, o algoritmo **descarta metade** do vetor. Portanto:

$$ T(n) = T(n/2) + c $$

Essa é uma recorrência clássica. Expandindo:

- 1ª chamada: `n`
- 2ª chamada: `n/2`
- 3ª chamada: `n/4`
- ...
- kª chamada: `n/(2^k) = 1`

Isso para quando `n/2^k = 1` → ou seja, $k = \log_2(n)$

Logo:

$$ T(n) = O(\log n) $$

Espaço – Pior Caso

Como a versão é **recursiva**, há consumo de pilha de chamadas. No pior caso (valor não está no vetor), a profundidade da recursão será:

$$ \text{Espaço} = O(\log n) $$

Resumo

- **Espaço (pior caso, versão recursiva)**: $\boxed{O(\log n)}$