DATA STRUCTURES

LAB SHEET 2

TITLE: Advanced Array and String Operations with Complexity Analysis

Objective

The objective of this assignment is to delve deeper into array and string data structures, perform advanced operations, and analyze their time and space complexities.

Problem Description

1. **Multi-dimensional Arrays:**
    - Implement a two-dimensional array and perform operations such as row-wise and column-wise insertion, deletion, and traversal.
    - Implement a program to find the transpose of a given matrix.
2. **Advanced String Operations:**
    - Implement string pattern matching algorithms (e.g., Knuth-Morris-Pratt algorithm).
    - Write a program to perform string compression using Run Length Encoding (RLE).
3. **Complexity Analysis:**
    - Analyze the time and space complexity of the multi-dimensional array operations and the string algorithms.
    - Provide best, average, and worst-case analyses for each implemented algorithm.

Instructions

1. **Multi-dimensional Array Implementation:**
    - Create a class TwoDimensionalArray with methods for row-wise and column-wise insertion, deletion, and traversal.
    - Write a function transposeMatrix that takes a matrix as input and returns its transpose.
2. **String Pattern Matching and Compression**:
    - Create a class StringAlgorithms with methods for the Knuth-Morris-Pratt pattern matching algorithm.
    - Write a function runLengthEncoding that takes a string as input and returns its RLE compressed form.
3. **Complexity Analysis**:
    - Write a report analyzing the time and space complexity of the implemented algorithms using Big O, Omega, and Theta notations.
    - Include best, average, and worst-case analyses for the algorithms.

CODE:

1. **Two Dimensional Array**

```
public class TwoDimensionalArray {
    // Private 2D array to store the matrix data
    private int[][] matrix;

    // Track current position for row and column insertions
    private int currentRow = 0;
    private int currentCol = 0;

    // Constructor: Initializes a 3x2 matrix filled with default values
    TwoDimensionalArray() {
        this.matrix = new int[3][2];
    }

    // Method to insert an entire row at the current row position
    public void insertRow(int[] row) {
        // Assigns the input row array to the current row in the matrix
        matrix[currentRow] = row;
        // Moves to the next row for any subsequent insertion
        currentRow += 1;
    }

    // Method to insert an entire column, filling the column sequentially in the
    // matrix
    public void insertColumn(int[] col) {
        for (int data : col) {
            // Adds each element of the input column array to the current matrix
position
            matrix[currentRow][currentCol] = data;
            // Moves to the next row for each element in the column
            currentRow += 1;
        }
    }

    // Method to delete (clear) all elements in a specified row by setting them to
0
    public void deleteRow(int index) {
        for (int i = 0; i < matrix[index].length; i++) {
            // Sets each element in the specified row to 0
            matrix[index][i] = 0;
        }
    }

    // Method to display the entire matrix, printing each element with a space
    public void display() {
        for (int[] row : matrix) {
```

```
            for (int data : row) {
                System.out.print(data + " ");
            }
            System.out.println(); // New line for each row
        }
    }

    // Method to traverse and print all non-zero elements in the matrix
    public void traverse() {
        for (int[] row : matrix) {
            for (int data : row) {
                if (data != 0) {
                    System.out.print(data + " ");
                }
            }
        }
    }

    // Main method to test the class functionality
    public static void main(String[] args) {
        // Create a new 3x2 matrix
        TwoDimensionalArray matrix = new TwoDimensionalArray();

        // Define a row to insert and add it to the matrix
        int[] row1 = { 1, 2 };
        matrix.insertRow(row1);

        // Define a column to insert and add it to the matrix
        int[] col1 = { 3, 4 };
        matrix.insertColumn(col1);

        // Delete the first row
        matrix.deleteRow(0);

        // Traverse and print all non-zero elements in the matrix
        matrix.traverse();
    }
}
```

2. **String Algorithms**

```java
import java.util.*;

public class StringAlgorithm {

    // Knuth-Morris-Pratt (KMP) pattern matching algorithm
    public int KMPSearch(String text, String pattern) {
        int[] lps = computeLPSArray(pattern);
        int n = text.length();
        int m = pattern.length();
        int i = 0, j = 0;

        while (i < n) {
            if (pattern.charAt(j) == text.charAt(i)) {
                j++;
                i++;
            }
            if (j == m) {
                return i - j; // pattern found at index (i - j)
            } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i++;
            }
        }
        return -1; // pattern not found
    }

    // Compute Longest Prefix Suffix (LPS) array for KMP algorithm
    public int[] computeLPSArray(String pattern) {
        int m = pattern.length();
        int[] lps = new int[m];
        int len = 0;
        int i = 1;

        while (i < m) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
```

```
            }
        }
        return lps;
    }

    // Run Length Encoding (RLE) string compression
    public String runLengthEncoding(String str) {
        StringBuilder encoded = new StringBuilder();
        int n = str.length();

        for (int i = 0; i < n; i++) {
            int count = 1;
            while (i < n - 1 && str.charAt(i) == str.charAt(i + 1)) {
                count++;
                i++;
            }
            encoded.append(count).append(str.charAt(i));
        }
        return encoded.toString();
    }

    public static void main(String[] args) {
        StringAlgorithm stringAlgo = new StringAlgorithm();

        // Test KMP Search
        String text = "abxabcabcaby";
        String pattern = "abcaby";
        int index = stringAlgo.KMPSearch(text, pattern);
        System.out.println("\nPattern found at index (KMP): " + index);

        // Test Run Length Encoding
        String str = "aaabbbcccaaa";
        String encodedStr = stringAlgo.runLengthEncoding(str);
        System.out.println("Run Length Encoding of \"" + str + "\": " +
encodedStr);
    }
}
```

Report: Advanced Array and String Operations with Complexity Analysis

Objective

The objective of this assignment is to explore advanced operations in array and string data structures and evaluate the time and space complexities for each implemented algorithm. The operations covered in this report include:

1. Multi-dimensional array manipulations (row/column insertion, deletion, traversal, and transpose).
2. Advanced string operations (Knuth-Morris-Pratt algorithm for pattern matching and Run Length Encoding for compression).
3. Complexity analysis of each algorithm with Big O, Omega, and Theta notations.

Problem Description

1. **Multi-dimensional Arrays**:
    - Implement a two-dimensional array supporting row/column insertion, deletion, and traversal.
    - Compute the transpose of a matrix.
2. **Advanced String Operations**:
    - Implement the Knuth-Morris-Pratt (KMP) algorithm for string pattern matching.
    - Implement Run Length Encoding (RLE) for string compression.
3. **Complexity Analysis**:
    - Analyze the time and space complexities for each algorithm and identify best, average, and worst-case scenarios.

Implementation

1. Multi-dimensional Arrays

Class: TwoDimensionalArray

- **Methods**:
    - insertRow: Inserts a new row at the end of the 2D array.
    - insertColumn: Inserts a new column at the end of the 2D array.
    - deleteRow: Deletes a specified row.
    - traverse: Traverses and displays the array.
    - transposeMatrix: Computes and returns the transpose of the array.

2. String Algorithms

Class: StringAlgorithms

- **Methods**:
    - KMPSearch: Implements the KMP pattern matching algorithm to search for a substring.
    - runLengthEncoding: Encodes a string using Run Length Encoding.

3.Complexity Analysis

Multi-dimensional Array Operations

1. **Insertion**:
    - **Time Complexity**: O(n)O(n)O(n) for row or column insertion.
    - **Space Complexity**: O(n×m)O(n \times m)O(n×m), where nnn is the number of rows and mmm is the number of columns.
2. **Deletion**:
    - **Time Complexity**: O(n)O(n)O(n) for row deletion.
    - **Space Complexity**: O(1)O(1)O(1), as no additional space is required.
3. **Transpose**:
    - **Time Complexity**: O(n×m)O(n \times m)O(n×m).
    - **Space Complexity**: O(n×m)O(n \times m)O(n×m) to store the transposed matrix.

String Algorithms

1. **Knuth-Morris-Pratt (KMP) Algorithm**:
    - **Time Complexity**: O(n+m)O(n + m)O(n+m), where nnn is the text length and mmm is the pattern length.
    - **Space Complexity**: O(m)O(m)O(m), for storing the LPS array.
2. **Run Length Encoding (RLE)**:
    - **Time Complexity**: O(n)O(n)O(n), where nnn is the length of the input string.
    - **Space Complexity**: O(n)O(n)O(n), for the output encoded string.

Test Cases

Multi-dimensional Array

Operation	Input	Expected Output
Insert Row and Column	insertRow({1, 2}), insertColumn({3, 4})	Updated matrix
Delete Row	deleteRow(0)	Matrix after deletion
Transpose Matrix	[[1, 2], [3, 4]]	[[1, 3], [2, 4]]

String Algorithms

Operation	Input	Expected Output
KMP Pattern Matching	"abxabcabcaby", "abcaby"	6
Run Length Encoding	"aaabbbcccaaa"	"3a3b3c3a"

Conclusion

The StringAlgorithms class provides efficient string manipulation algorithms. The KMP algorithm significantly improves the time complexity of string matching, and both the LPS array and RLE techniques optimize space and time when handling large strings.

The TwoDimensionalArray class provides a variety of operations for managing a 2D array, including insertion, removal, traversal, and matrix transposition. The time complexity for most operations depends on the size of the array, especially when adding or removing rows/columns. The space complexity is typically dominated by the size of the array being manipulated.