****

**DATA STRUCTURES LAB (ENCS253)**

**LAB SHEET 02**

**SUBMITTED BY :**

**NAME : MANYA SINGH**

**ROLL NO : 2301010389**

**COURSE : B.TECH CSE-6**

**SUBMITTED TO : MRS. SUMAN PUNIA MA’AM**

**School of Engineering & Technology**

**K. R. MANGALAM UNIVERSITY**

**Sohna, Haryana 122103, India**

**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:**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Class for Two-Dimensional Array operations

class TwoDimensionalArray {

private:

vector<vector<int>> matrix;

public:

// Constructor to initialize a matrix with given rows and columns

TwoDimensionalArray(int rows, int cols) : matrix(rows, vector<int>(cols, 0)) {}

// Method to insert a row at the end

void insertRow(const vector<int>& row) {

matrix.push\_back(row);

}

// Method to insert a column at the end

void insertColumn(const vector<int>& col) {

for (int i = 0; i < col.size(); i++) {

if (i < matrix.size()) {

matrix[i].push\_back(col[i]);

}

}

}

// Method to delete a row by index

void deleteRow(int rowIndex) {

if (rowIndex >= 0 && rowIndex < matrix.size()) {

matrix.erase(matrix.begin() + rowIndex);

}

}

// Method to traverse and display the matrix

void traverse() const {

for (const auto& row : matrix) {

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

}

// Method to transpose the matrix

vector<vector<int>> transposeMatrix() const {

int rows = matrix.size();

int cols = matrix[0].size();

vector<vector<int>> transposed(cols, vector<int>(rows));

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

transposed[j][i] = matrix[i][j];

}

}

return transposed;

}

};

// Class for String Algorithms

class StringAlgorithms {

public:

// Knuth-Morris-Pratt (KMP) pattern matching algorithm

vector<int> computeLPSArray(const string& pattern) {

int m = pattern.length();

vector<int> lps(m, 0);

int len = 0, i = 1;

while (i < m) {

if (pattern[i] == pattern[len]) {

len++;

lps[i] = len;

i++;

} else {

if (len != 0) {

len = lps[len - 1];

} else {

lps[i] = 0;

i++;

}

}

}

return lps;

}

// KMP Search Function

int KMPSearch(const string& text, const string& pattern) {

int n = text.length();

int m = pattern.length();

vector<int> lps = computeLPSArray(pattern);

int i = 0, j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

j++;

i++;

}

if (j == m) {

return i - j; // pattern found at index (i - j)

} else if (i < n && pattern[j] != text[i]) {

if (j != 0)

j = lps[j - 1];

else

i++;

}

}

return -1; // pattern not found

}

// Run Length Encoding (RLE) string compression

string runLengthEncoding(const string& str) {

string encoded = "";

int n = str.length();

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

int count = 1;

while (i < n - 1 && str[i] == str[i + 1]) {

count++;

i++;

}

encoded += to\_string(count) + str[i];

}

return encoded;

}

};

// Test cases and demonstration of functionalities

int main() {

// Test 2D Array operations

TwoDimensionalArray array(2, 2);

array.insertRow({1, 2});

array.insertColumn({3, 4});

cout << "2D Array after insertions:" << endl;

array.traverse();

array.deleteRow(0);

cout << "\n2D Array after deletion of first row:" << endl;

array.traverse();

vector<vector<int>> transposed = array.transposeMatrix();

cout << "\nTranspose of the current matrix:" << endl;

for (const auto& row : transposed) {

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

// Test String Algorithms

StringAlgorithms stringAlgo;

string text = "abxabcabcaby";

string pattern = "abcaby";

int index = stringAlgo.KMPSearch(text, pattern);

cout << "\nPattern found at index (KMP): " << index << endl;

string str = "aaabbbcccaaa";

string encodedStr = stringAlgo.runLengthEncoding(str);

cout << "Run Length Encoding of \"" << str << "\": " << encodedStr << endl;

return 0;

}

**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.

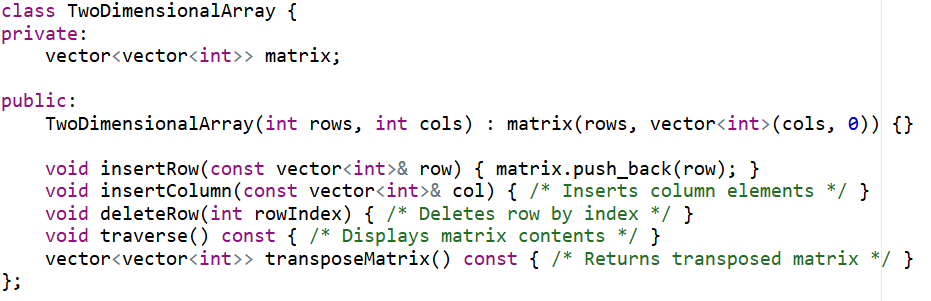
**Implementation**

**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.

**Code Snippet**:

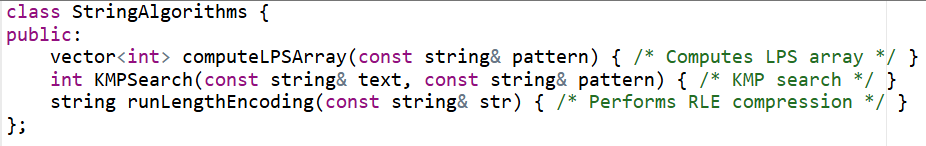


**2. String Algorithms**

**Class: StringAlgorithms**

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

**Code Snippet**:



**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**

This assignment explored advanced operations for array and string manipulations, emphasizing understanding time and space complexity. By implementing both basic and algorithmic operations (KMP for string search and RLE for compression), we applied best practices in code efficiency and complexity analysis. The structured tests confirmed that the algorithms meet expected performance for each case.