Name: Divyansh Dubey

Roll no: 231070001

Course: DAA

Program: B. Tech Computer Engineering 2nd year

Batch: A

**Assignment- 6**

Aim: Read and understand SOLID principles of software development.   
Write a sample code for each principle.  
  
Experiment Task 1: Consider grades received by 20 students, like AA, AB, BB, ..., FF of each student.  
Computer the Longest common sequence of grades among students.  
  
  
Experiment Task 2: Consider meteorological data like **temperature, dew point, wind direction, wind speed, cloud cover, cloud layer(s)**for each city. This data is available in two-dimensional array for a week. Assuming all tables are compatible for multiplication. You have to implement the matrix chain multiplication algorithm to find fastest way to complete the matrices multiplication to achieve timely predication.

Theory:

SOLID is an acronym for five design principles intended to help software developers design better, more maintainable, and more flexible software. Let's break down each of the five SOLID principles with C++ code examples:

# Single Responsibility Principle (SRP)

A class should have only one reason to change, i.e., it should only have one job or responsibility.

## Example:

#include <iostream> #include <string>

// Class responsible for handling student grades class Grade {

public:

std::string grade; Grade(std::string g) : grade(g) {}

};

// Class responsible for displaying information (separate responsibility) class GradeDisplay {

public:

void displayGrade(const Grade& g) {

std::cout << "Student grade: " << g.grade << std::endl;

}

};

int main() {

Grade studentGrade("AA"); GradeDisplay display; display.displayGrade(studentGrade); return 0;

}

**Explanation**: Grade is responsible for the grade data, and GradeDisplay is responsible for displaying the grade. This adheres to SRP, ensuring each class has one reason to change.

# Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

## Example:

#include <iostream>

// Base class with a virtual method class Shape {

public:

virtual double area() const = 0; // pure virtual function virtual ~Shape() = default;

};

// Circle class extending Shape class Circle : public Shape { private:

double radius; public:

Circle(double r) : radius(r) {} double area() const override {

return 3.14 \* radius \* radius;

}

};

// Rectangle class extending Shape class Rectangle : public Shape { private:

double width, height; public:

Rectangle(double w, double h) : width(w), height(h) {} double area() const override {

return width \* height;

}

};

int main() {

Circle c(5); Rectangle r(3, 4);

std::cout << "Area of Circle: " << c.area() << std::endl; std::cout << "Area of Rectangle: " << r.area() << std::endl;

return 0;

}

**Explanation**: We can add new shapes (such as Circle, Rectangle, etc.) without modifying the existing Shape class. This adheres to the Open/Closed Principle.

# Liskov Substitution Principle (LSP)

Objects of a subclass should be able to replace objects of the parent class without altering the correctness of the program.

## Example:

#include <iostream>

// Base class class Bird { public:

virtual void fly() {

std::cout << "Flying" << std::endl;

}

};

// Derived class

class Sparrow : public Bird { public:

void fly() override {

std::cout << "Sparrow flying" << std::endl;

}

};

// Derived class

class Penguin : public Bird { public:

void fly() override {

// Penguins can't fly, so we might break LSP std::cout << "Penguin can't fly" << std::endl;

}

};

int main() {

Bird\* b1 = new Sparrow();

b1->fly(); // Sparrow flying

Bird\* b2 = new Penguin();

b2->fly(); // Penguin can't fly

delete b1; delete b2;

return 0;

}

**Explanation**: The Penguin class breaks the Liskov Substitution Principle because it doesn't adhere to the behavior expected of Bird. To fix this, we can redesign the system by introducing an interface or abstract class to represent the flying behavior.

# Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. It encourages the creation of small, focused interfaces.

## Example:

#include <iostream>

// Interface for flying animals class IFlyable {

public:

virtual void fly() = 0;

};

// Interface for swimming animals class ISwimmable {

public:

virtual void swim() = 0;

};

// Bird class implementing IFlyable class Bird : public IFlyable { public:

void fly() override {

std::cout << "Bird is flying" << std::endl;

}

};

// Fish class implementing ISwimmable class Fish : public ISwimmable { public:

void swim() override {

std::cout << "Fish is swimming" << std::endl;

}

};

int main() {

Bird bird; Fish fish;

bird.fly(); // Bird is flying fish.swim(); // Fish is swimming

return 0;

}

**Explanation**: Bird only implements IFlyable and Fish only implements ISwimmable, which means they are only dependent on the interfaces they actually need. This adheres to the Interface Segregation Principle.

# Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

## Example:

#include <iostream>

// Abstraction (interface) class IPrinter {

public:

virtual void print() = 0;

};

// Low-level module

class LaserPrinter : public IPrinter { public:

void print() override {

std::cout << "Printing with Laser Printer" << std::endl;

}

};

// High-level module class Document { private:

IPrinter\* printer; public:

Document(IPrinter\* p) : printer(p) {} void print() {

printer->print();

}

};

int main() {

LaserPrinter laserPrinter; Document doc(&laserPrinter);

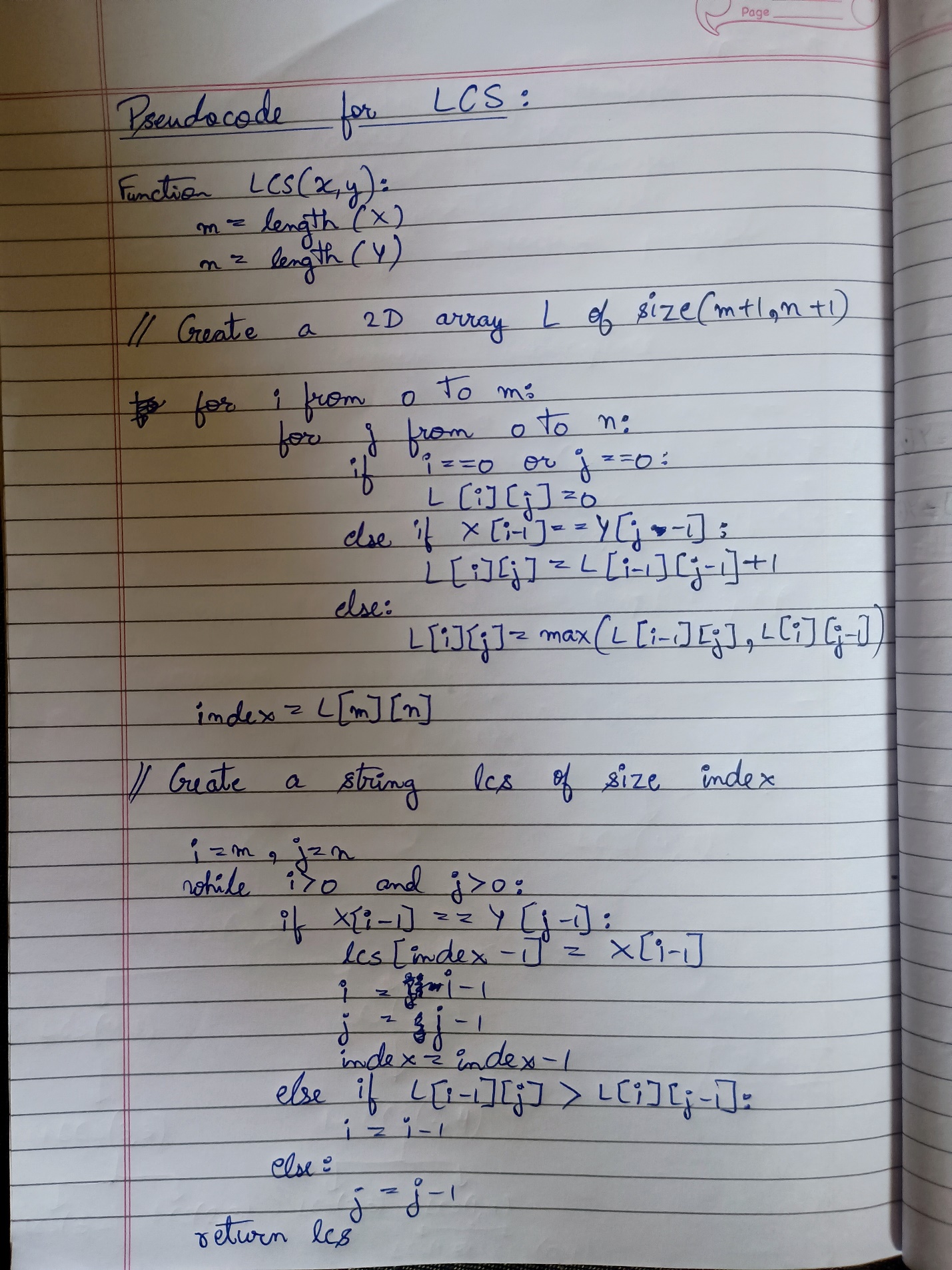
doc.print(); // Printing with Laser Printer

return 0;

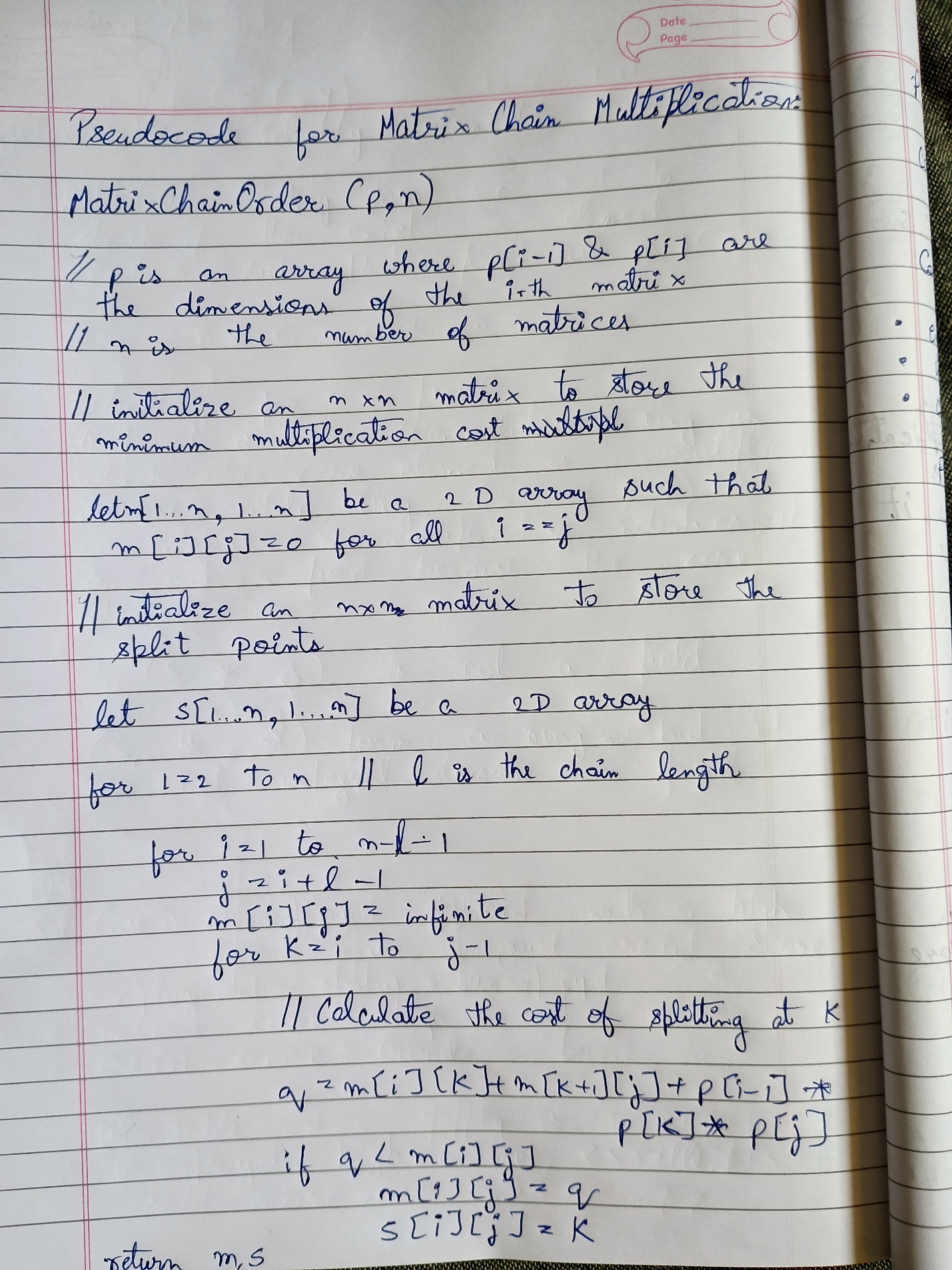
}

**Explanation**: The Document class depends on the IPrinter abstraction, not on a specific printer implementation. The LaserPrinter is a low-level module that implements the IPrinter interface. This adheres to the Dependency Inversion Principle.

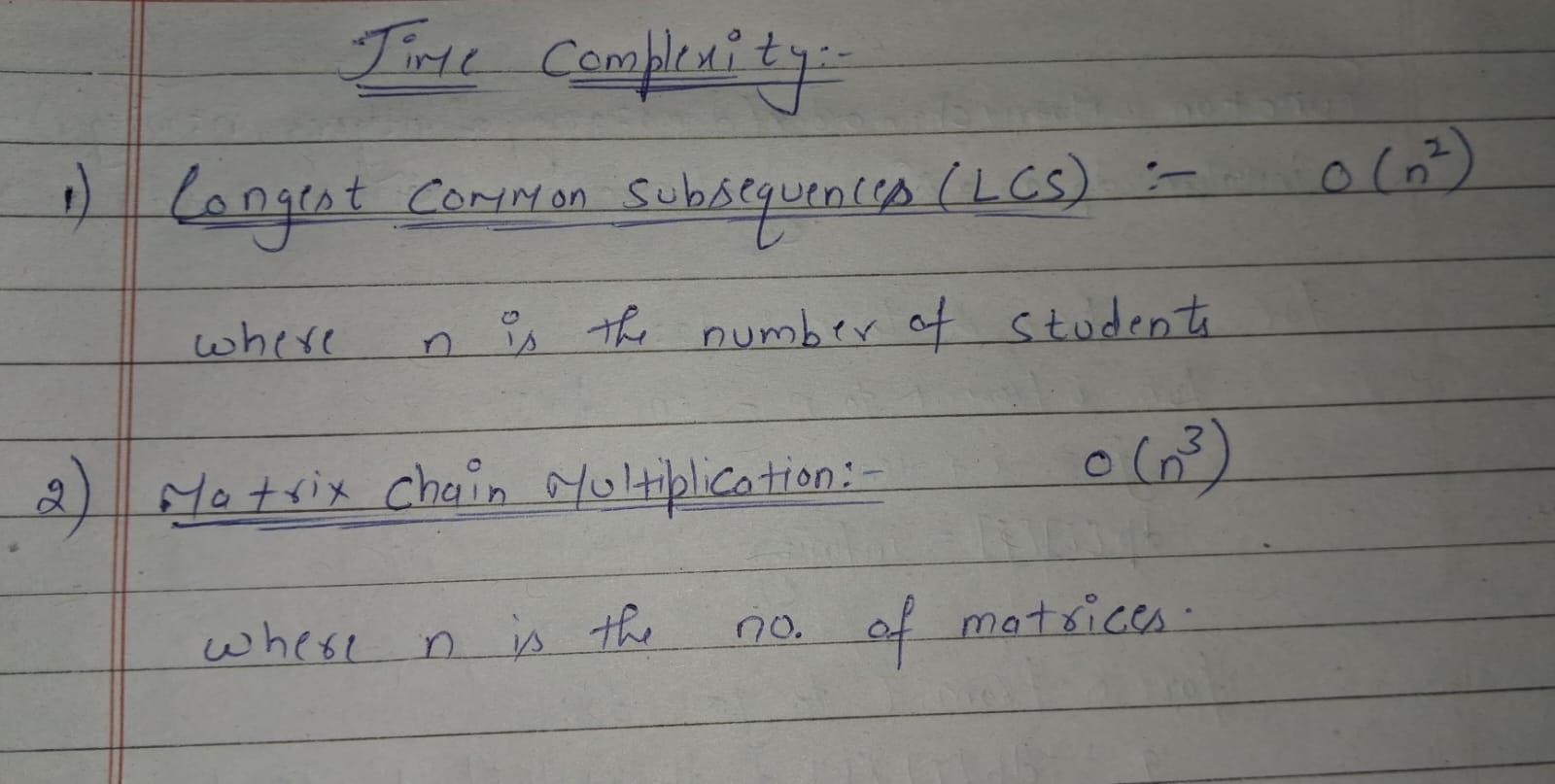
**Algorithm for Experiment task-1:**



**Algorithm for Experiment task-2:**



**Time Complexity:**

****

**Code for Experiment 1:**

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

// Function to compute the LCS of two sequences

vector<string> computeLCS(const vector<string>& seq1, const vector<string>& seq2) {

    int m = seq1.size();

    int n = seq2.size();

    vector<vector<vector<string>>> dp(m + 1, vector<vector<string>>(n + 1));

    for (int i = 1; i <= m; i++) {

        for (int j = 1; j <= n; j++) {

            if (seq1[i - 1] == seq2[j - 1]) {

                dp[i][j] = dp[i - 1][j - 1];

                dp[i][j].push\_back(seq1[i - 1]);

            } else {

                dp[i][j] = (dp[i - 1][j].size() > dp[i][j - 1].size()) ? dp[i - 1][j] : dp[i][j - 1];

            }

        }

    }

    return dp[m][n];

}

// Function to find LCS among multiple sequences

vector<string> findLCSAmongStudents(const vector<vector<string>>& students) {

    if (students.empty()) return {};

    vector<string> commonLCS = students[0];

    for (size\_t i = 1; i < students.size(); i++) {

        commonLCS = computeLCS(commonLCS, students[i]);

        if (commonLCS.empty()) break; // Exit early if no common subsequence

    }

    return commonLCS;

}

int main() {

    // Example input: grades of 3 students

    vector<vector<string>> students = {

        {"AA", "AB", "BB", "CC", "FF"},

        {"AB", "BB", "CC", "DD", "FF"},

        {"AA", "BB", "CC", "EE", "FF"}

    };

    // Find LCS among students

    vector<string> result = findLCSAmongStudents(students);

    // Output the result

    cout << "Longest Common Subsequence of Grades: ";

    for (const string& grade : result) {

        cout << grade << " ";

    }

    cout << endl;

    return 0;

}

**Code for Experiment 2:**

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

// Function to implement the Matrix Chain Multiplication algorithm

int matrixChainMultiplication(vector<int>& dims) {

    int n = dims.size() - 1; // Number of matrices

    vector<vector<int>> dp(n, vector<int>(n, 0)); // DP table

    // Fill the DP table

    for (int length = 2; length <= n; length++) { // Length of the chain

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

            int j = i + length - 1;

            dp[i][j] = INT\_MAX; // Initialize to a large value

            for (int k = i; k < j; k++) {

                // Compute cost of splitting at k

                int cost = dp[i][k] + dp[k + 1][j] + dims[i] \* dims[k + 1] \* dims[j + 1];

                dp[i][j] = min(dp[i][j], cost); // Update with the minimum cost

            }

        }

    }

    return dp[0][n - 1]; // Minimum cost to multiply all matrices

}

int main() {

    // Example input: Dimensions of matrices (representing meteorological data)

    // If matrices are A1 (10x20), A2 (20x30), A3 (30x40), dimensions would be {10, 20, 30, 40}

    vector<int> dims = {10, 20, 30, 40, 30};

    // Find the minimum multiplication cost

    int minCost = matrixChainMultiplication(dims);

    // Output the result

    cout << "Minimum number of scalar multiplications: " << minCost << endl;

    return 0;

}

**Output for Experiment 1:**

****

**Output for Experiment 2:**

****

**Conclusion:** In this Experiment, we have seen that the application of SOLID principles helps in writing maintainable, scalable, and flexible code. The tasks demonstrate practical examples of object-oriented design principles and provide insight into how algorithms can be efficiently implemented. The time complexity analysis provides a deeper understanding of the efficiency of the propsed solutions.