DAA LAB 6

Amit Rajkumar Ingle

S.Y.B.Tech computer

231070020

TASK-1

AIM: Read and understand SOLID principles of software development. Write a sample code for each principle.

- Solid principles :

1.Single Responsibility Principle (SRP) :

Each class should have only one reason to change, meaning it should have only one responsibility or task.

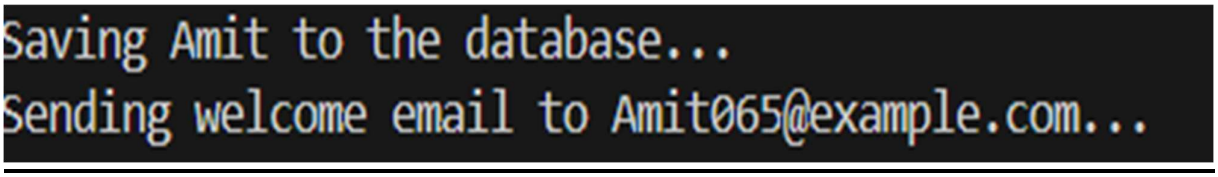
Imagine a User class that handles both data storage (saving to a database) and email communication (sending a welcome email). According to SRP, these are two distinct responsibilities: data handling and email communication. By splitting them into separate classes (e.g., UserDatabase for storage and EmailService for communication), each class now has a single, clear responsibility, which makes them easier to maintain and reuse.

Implementation :

```
#include <iostream>
#include <string>
using namespace std;
```

```
// Class representing user data
class User {
public:
    string name;
    string email;

    User(string n, string e) : name(n), email(e) {}
};

// Class responsible for database operations related to User
class UserDatabase {
public:
    void save(const User& user) {
        cout << "Saving " << user.name << " to the database...\n";
    }
};

// Class responsible for email operations related to User
class EmailService {
public:
    void sendWelcomeEmail(const User& user) {
        cout << "Sending welcome email to " << user.email <<
"...\n";
    }
};

// Main function to demonstrate SRP-compliant design
int main() {
    // Create a new user
    User user("Amit", "Amit065@example.com");

    // Instantiate services
    UserDatabase;
    EmailService;

    // Save user to the database and send a welcome email
    userDatabase.save(user);
    emailService.sendWelcomeEmail(user);

    return 0;
}
```

OUTPUT:



2.Open/Closed Principle (OCP) :

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

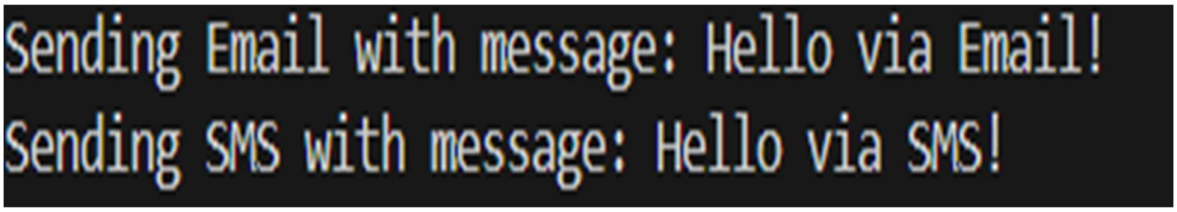
Suppose we have a notification system with a Notification base class. We want the system to be open to new notification types like SMS, push notifications, or emails without modifying the original Notification class. By defining a base Notification class and implementing subclasses like EmailNotification and SMSNotification, we add new types of notifications without altering the existing ones. This design is scalable and minimizes code alterations, thereby maintaining stability.

Implementation :

```
#include <iostream>
#include <string>
using namespace std;

// Base class for Notification
class Notification {
public:
    virtual void send(string message) = 0; // Pure virtual function
};

class EmailNotification : public Notification {
public:
    void send(string message) override {
        cout << "Sending Email with message: " << message << endl;
    }
};
```

```
class SMSNotification : public Notification {
public:
    void send(string message) override {
        cout << "Sending SMS with message: " << message << endl;
    }
};

// Client code
void notify(Notification* notification, string message) {
    notification->send(message);
}

int main() {
    EmailNotification email;
    SMSNotification sms;

    notify(&email, "Hello via Email!");
    notify(&sms, "Hello via SMS!");

    return 0;
}
```

OUTPUT:



3.Liskov Substitution Principle (LSP) :

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Imagine we have a Bird class with a fly method. If we create a Sparrow class that can fly, it's valid to replace Bird with Sparrow. However, if we add an Ostrich class that cannot fly but still inherits from Bird, it breaks LSP because it changes the expected behaviour of fly. By introducing a FlyingBird subclass for birds that can fly, we can separate the behaviours, maintaining LSP compliance since each subclass now has a behaviour that is predictable and consistent with its type.

implementation:

```
#include <iostream>
using namespace std;

// Base class
class Bird {
public:
    virtual void fly() {
        cout << "I can fly!" << endl;
    }
};

// Subclass which can fly
class Sparrow : public Bird {};

// Subclass which cannot fly (breaks LSP if used with Bird pointer)
class Ostrich : public Bird {
public:
    void fly() override {
        throw "Cannot fly!";
    }
};
```

```
// Correct implementation (LSP compliant)
class Bird {
public:
    virtual void fly() = 0; // Pure virtual
};

class FlyingBird : public Bird {
public:
    void fly() override {
        cout << "I can fly!" << endl;
    }
};

class Sparrow : public FlyingBird {};

class Ostrich : public Bird {
public:
    void fly() override {
        cout << "I cannot fly!" << endl;
    }
};

int main() {
    Sparrow;
    Ostrich;

    Bird* bird1 = &sparrow;
    Bird* bird2 = &ostrich;

    bird1->fly(); // Works fine
    bird2->fly(); // Works fine, no exception

    return 0;
}
```

OUTPUT:

4.Interface Segregation Principle (ISP) :

Clients should not be forced to depend on interfaces they do not use.

Consider a Machine interface that includes print, scan, and fax methods. A class that only needs print capabilities would still be forced to implement scan and fax methods, violating ISP. By breaking down machines into separate Printer, Scanner, and Fax interfaces, classes can now implement only the interfaces they require, avoiding irrelevant dependencies and simplifying the design.

Implementation:

```
#include <iostream>
#include <string>
using namespace std;

// Violates ISP - a single interface with unrelated methods
class Machine {
public:
    virtual void print(string document) = 0;
    virtual void scan(string document) = 0;
    virtual void fax(string document) = 0;
};

// Correct implementation with separate interfaces
class Printer {
public:
    virtual void print(string document) = 0;
};

class Scanner {
public:
    virtual void scan(string document) = 0;
};

class Fax {
public:
    virtual void fax(string document) = 0;
```

```
};

// Implementing specific functionalities
class MultiFunctionPrinter : public Printer, public Scanner, public
Fax {
public:
    void print(string document) override {
        cout << "Printing: " << document << endl;
    }
    void scan(string document) override {
        cout << "Scanning: " << document << endl;
    }
    void fax(string document) override {
        cout << "Faxing: " << document << endl;
    }
};

class SimplePrinter : public Printer {
public:
    void print(string document) override {
        cout << "Printing: " << document << endl;
    }
};
```

OUTPUT:

```
Printing: Document1
Scanning: Document1
Faxing: Document1
Printing: Document2
Scanning: Document3
```

5.Dependency Inversion Principle (DIP) :

High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

Let's say we have a DataAccess class responsible for retrieving data. Instead of directly depending on a specific database implementation like MySQLDatabase, which is a low-level detail, we introduce an abstract Database interface that both MySQLDatabase and PostgreSQLDatabase can implement. The DataAccess class then depends on Database, an abstraction, instead of a concrete database class. This makes it easy to swap out or add different database implementations without modifying DataAccess, maintaining a flexible and extensible design.

Implementation:

```
#include <iostream>
using namespace std;

// Abstraction
class Database {
public:
    virtual void connect() = 0;
};

// Low-level module
class MySQLDatabase : public Database {
public:
    void connect() override {
        cout << "Connecting to MySQL Database" << endl;
    }
};

class PostgreSQLDatabase : public Database {
public:
    void connect() override {
```

```
        cout << "Connecting to PostgreSQL Database" << endl;
    }
};

// High-level module
class DataAccess {
private:
    Database* database;
public:
    DataAccess(Database* db) : database(db) {}

    void getData() {
        database->connect();
        cout << "Fetching data" << endl;
    }
};

int main() {
    MySQLDatabase mysqlDb;
    PostgreSQLDatabase postgresDb;

    DataAccess dataAccess1(&mysqlDb);
    DataAccess dataAccess2(&postgresDb);

    dataAccess1.getData();
    dataAccess2.getData();

    return 0;
}
```

OUTPUT:

```
Connecting to MySQL Database
Fetching data
Connecting to PostgreSQL Database
Fetching data
```

Experiment Task 1

AIM: Consider grades received by 20 students, like AA, AB, BB, ..., FF of each student.

Computer the Longest common sequence of grades among students.

THEORY:

The Longest Common Subsequence (LCS) problem is used to find the longest sequence that appears in the same order across multiple grade sequences, though not necessarily in continuous segments. Here, each student's grade sequence—denoted by symbols like AA, AB, BB—serves as a distinct series, and our goal is to identify the longest common subsequence that appears across these sequences. For instance, if two students have grade sequences "AAAB" and "ABAB," their LCS would be "AAB." This analysis is helpful for identifying consistent patterns in academic performance, revealing areas where students may have shared strengths or challenges.

By finding common subsequences in grades, educators can spot recurring trends that indicate the impact of certain lessons or assignments on student performance. For example, if a pattern like "BB, BC, BB" frequently appears across students, it may signal a shared area of difficulty, while consistently high sequences could point to effective teaching strategies.

To compute the LCS efficiently, we use a Dynamic Programming approach. By constructing a two-dimensional table, the algorithm systematically compares characters across sequences, building up the LCS length based on matches and ignoring mismatches. The result reveals the longest shared sequence between students' grades, helping educators understand collective performance trends and make data-driven decisions to support student learning.

PROGRAM:

LCS GENERATOR:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <string>
#include <vector>


// Utility function to get a random grade from the grade pool
std::string getRandomGrade(const std::vector<std::string>&
gradePool) {
    int index = std::rand() % gradePool.size();
    return gradePool[index];
}


// Function to generate a CSV file with student grades and handle
edge cases
void generateStudentGradesCSV(const std::string& filename, int
numStudents, bool isEmpty, bool includeInvalidGrades, bool
includeIncorrectLength, bool includeMissingData) {
```

```
    // Check if an empty CSV file should be created based on the
flag
    if (isEmpty) {
        std::ofstream file(filename);
        file << "Student ID,Grade\n"; // Only header
        file.close();
        return; // Exit after creating an empty file
    }

    // Expanded grade pool to allow variety in longer sequences
    std::vector<std::string> gradePool = {"AA", "AB", "BB", "BC",
"CC", "CD", "DD", "FF"};
    std::ofstream file(filename);
    file << "Student ID,Grade\n"; // Write CSV header

    for (int i = 1; i <= numStudents; ++i) {
        std::string studentID = "student" + std::to_string(i);

        // Generate a random grades string for the student
        std::string grades;
        if (includeIncorrectLength && (std::rand() % 2 == 0)) {
            // Randomly generate either fewer or more than 15 grades
            int length = (std::rand() % 2 == 0) ? 10 : 20;  //
Either fewer or more grades
            for (int j = 0; j < length; ++j) {
                grades += getRandomGrade(gradePool);
            }
        } else {
            // Generate exactly 15 grades
            for (int j = 0; j < 15; ++j) {
                grades += getRandomGrade(gradePool);
```

```
            }
        }

        // Introduce invalid grades if the flag is enabled
        if (includeInvalidGrades && (std::rand() % 2 == 0)) {
            grades[std::rand() % grades.size()] = '$';  // Insert an
invalid character
        }

        // Introduce missing data if the flag is enabled
        if (includeMissingData && (std::rand() % 2 == 0)) {
            grades = "";  // Make the grade string empty to simulate
missing data
        }

        // Write data to CSV
        file << studentID << "," << (grades.empty() ? "N/A" :
grades) << "\n";
    }

    file.close();
}

int main() {
    // Seed the random number generator with the current time
    std::srand(static_cast<unsigned>(std::time(nullptr)));

    // Use a simple filename without specifying a path (file will be
created in the current directory)
    std::string file_path = "students_grades8.csv";  // This will
create the file in the current directory
```

```
    // Flags for edge cases
    bool isEmpty = false;               // Generate a non-empty file (set to false)
    bool includeInvalidGrades = false;    // Include invalid grades (e.g., special characters)
    bool includeIncorrectLength = false;  // Include entries with incorrect grade length
    bool includeMissingData = false;      // Include missing data

    // Generate a CSV file with the specified number of students and edge cases
    generateStudentGradesCSV(file_path, 20, isEmpty, includeInvalidGrades, includeIncorrectLength, includeMissingData);

    std::cout << "CSV file " << file_path << " generated successfully.\n";
    return 0;
}
```

LCS SOLVER:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <unordered_set>
#include <algorithm>

// Constants remain the same
const int EXPECTED_COURSES = 15;
const int GRADE_LENGTH = EXPECTED_COURSES * 2;
const std::unordered_set<std::string> VALID_GRADES = {"AA", "AB", "BB", "BC", "CC", "CD", "DD", "FF"};
```

```
class GradeValidator {
public:
    static bool isValidGradeString(const std::string& grades) {
        if (grades.empty() || grades.length() != GRADE_LENGTH) {
            return false;
        }

        for (size_t i = 0; i < grades.length(); i += 2) {
            std::string grade = grades.substr(i, 2);
            if (VALID_GRADES.find(grade) == VALID_GRADES.end()) {
                return false;
            }
        }
        return true;
    }
};

class GradeReader {
public:
    static std::vector<std::string> readValidGrades(const
std::string& filename) {
        std::vector<std::string> validGrades;
        std::ifstream file(filename);

        if (!file.is_open()) {
            std::cerr << "Error: Unable to open file " << filename
<< std::endl;
            return validGrades;
        }

        std::string line;
```

```
        std::getline(file, line); // Skip header


        while (std::getline(file, line)) {
            size_t commaPos = line.find(',');
            if (commaPos != std::string::npos) {
                std::string grades = line.substr(commaPos + 1);
                if (GradeValidator::isValidGradeString(grades)) {
                    validGrades.push_back(grades);
                } else {
                    std::cerr << "Invalid grade string found: " <<
grades << std::endl;
                }
            } else {
                std::cerr << "Invalid line format: " << line <<
std::endl;
            }
        }


        if (validGrades.empty()) {
            std::cerr << "Warning: No valid grades found in the
file" << std::endl;
        } else {
            std::cout << "Successfully processed " <<
validGrades.size() << " valid grade entries" << std::endl;
        }


        return validGrades;
    }
};


class GeneralizedLCSFinder {
```

```
private:
    // Helper function to check if a character exists at position i
in all strings
    static bool isCommonAtPosition(const std::vector<std::string>&
strings,
                                   const std::vector<int>& positions,
                                   char ch) {
        for (size_t i = 0; i < strings.size(); i++) {
            if (positions[i] >= strings[i].length() ||
                strings[i][positions[i]] != ch) {
                return false;
            }
        }
        return true;
    }

    // Helper function to find all possible LCS recursively
    static void findAllLCSHelper(const std::vector<std::string>&
strings,
                                 std::vector<int>& positions,
                                 std::string& current,
                                 std::string& best,
                                 std::unordered_set<std::string>&
solutions) {
        // Try each character from the first string as potential
next character
        bool found = false;
        const std::string& firstStr = strings[0];  // Changed to
const reference

        for (int i = positions[0]; i < firstStr.length(); i++) {
            char ch = firstStr[i];
```

```
            std::vector<int> newPositions = positions;
            bool isCommon = true;

            // Check if this character exists at some position in
all other strings
            for (size_t j = 1; j < strings.size(); j++) {
                bool charFound = false;
                for (int k = positions[j]; k < strings[j].length();
k++) {
                    if (strings[j][k] == ch) {
                        newPositions[j] = k + 1;
                        charFound = true;
                        break;
                    }
                }
                if (!charFound) {
                    isCommon = false;
                    break;
                }
            }

            if (isCommon) {
                found = true;
                newPositions[0] = i + 1;
                current.push_back(ch);
                findAllLCSHelper(strings, newPositions, current,
best, solutions);
                current.pop_back();
            }
        }
```

```
        if (!found && current.length() >= best.length()) {
            if (current.length() > best.length()) {
                best = current;
                solutions.clear();
            }
            solutions.insert(current);
        }
    }

public:
    static std::string findLCS(const std::vector<std::string>&
grades) {
        if (grades.empty()) return "";
        if (grades.size() == 1) return grades[0];

        // Initialize positions vector
        std::vector<int> positions(grades.size(), 0);
        std::string current;
        std::string best;
        std::unordered_set<std::string> solutions;

        // Find all possible LCS
        findAllLCSHelper(grades, positions, current, best,
solutions);

        // If multiple solutions exist with same length, choose
lexicographically smallest
        if (!solutions.empty()) {
            return *std::min_element(solutions.begin(),
solutions.end());
        }
```

```
        return "";
    }
};
class BruteForceLCSFinder {
private:
    // Helper function to generate all subsequences of a string
    static void generateSubsequences(const std::string& str, int
index, std::string current, std::unordered_set<std::string>&
subsequences) {
        if (index == str.size()) {
            subsequences.insert(current);
            return;
        }

        // Include the current character
        generateSubsequences(str, index + 1, current + str[index],
subsequences);

        // Exclude the current character
        generateSubsequences(str, index + 1, current, subsequences);
    }

public:
    // Function to find the longest common subsequence using brute
force
    static std::string findLCS(const std::vector<std::string>&
strings) {
        if (strings.empty()) return "";

        // Generate all subsequences of the first string
        std::unordered_set<std::string> commonSubsequences;
        generateSubsequences(strings[0], 0, "", commonSubsequences);
```

```
        // For each other string, filter only the common
subsequences
        for (size_t i = 1; i < strings.size(); i++) {
            std::unordered_set<std::string> currentSubsequences;
            generateSubsequences(strings[i], 0, "",
currentSubsequences);

            // Retain only the subsequences that are common
            for (auto it = commonSubsequences.begin(); it !=
commonSubsequences.end();) {
                if (currentSubsequences.find(*it) ==
currentSubsequences.end()) {
                    it = commonSubsequences.erase(it);
                } else {
                    ++it;
                }
            }
        }

        // Find the longest subsequence among the common ones
        std::string longest = "";
        for (const std::string& subseq : commonSubsequences) {
            if (subseq.length() > longest.length() ||
                (subseq.length() == longest.length() && subseq <
longest)) {
                longest = subseq;
            }
        }

        return longest;
    }
```

```
};
int main() {
    std::string filename = "students_grades8.csv";

    // Read and validate grades
    std::vector<std::string> validGrades =
GradeReader::readValidGrades(filename);

    if (validGrades.empty()) {
        std::cout << "No valid grade data found.\n";
        return 1;
    }

    // Find and display GLCS
    std::string glcs = GeneralizedLCSFinder::findLCS(validGrades);

    // Output results
    std::cout << "\nResults:" << std::endl;
    std::cout << "Number of valid grade entries: " <<
validGrades.size() << std::endl;
    std::cout << "GLCS Length: " << glcs.length() << std::endl;
    std::cout << "GLCS: " << glcs << std::endl;

    // Print valid grades for verification
    std::cout << "\nValid Grades Processed:" << std::endl;
    for (size_t i = 0; i < validGrades.size(); i++) {
        std::cout << "Entry " << (i+1) << ": " << validGrades[i] <<
std::endl;
    }
    return 0;
}
```

OUTPUT:

```
Successfully processed 20 valid grade entries

Results:
Number of valid grade entries: 20
GLCS Length: 3
GLCS: ABA
```

```
Successfully processed 20 valid grade entries

Results:
Number of valid grade entries: 20
GLCS Length: 4
GLCS: BBBB
```

```
Successfully processed 16 valid grade entries

Results:
Number of valid grade entries: 16
GLCS Length: 5
GLCS: BBBBC

Valid Grades Processed:
Entry 1: FFBBBCCDAACDCCFFCCDDBCCCFFBCDD
Entry 2: BBBCDDBBABAAFFDDFFDDAADDBCFFFF
Entry 3: BBAAFFBBDDCDABFFABBCCCAAFFDDFF
Entry 4: CCDDABAAAACDBCAAFFCCAABCBCDDCC
Entry 5: DDBCCCCDCCCDCCABBBDDBBCDABAACD
Entry 6: FFCCBCDDFFBCAACCBBAAABBCFFAACD
Entry 7: FFDDBCBCBCCCFFCCFFCCBBDDABBCFF
Entry 8: DDBCBCCCCDDDBCABFFCDCCBCAACDFF
Entry 9: FFDDABAAABCDBBDDDDBBDDFFBCBBCC
Entry 10: ABAAAACDBBBCAADDDDCDBBCCCDBCBB
Entry 11: FFBCAADDBBCCABAAABFFABCDCCBBCD
Entry 12: CCCCFFBCBCCCCDFFFFBBBBDDBBCDAA
Entry 13: FFBBAACDDDFFAAFFBCCDBCAABBBCBB
Entry 14: FFCCBCABFFDDBCCCBCABBCCCBBBBAB
Entry 15: CCAACDFFCDBBBBBCAACDCDABDDCCFF
Entry 16: ABDDBBDDDDABCDCCABBCCDCCCCCCFF
```

```
Successfully processed 5 valid grade entries

Results:
Number of valid grade entries: 5
GLCS Length: 8
GLCS: BBCDDBBC

Valid Grades Processed:
Entry 1: FFDDBCBCBCCCFFCCFFCCBBDDABBCFF
Entry 2: DDBCBCCCCDDDBCABFFCDCCBCAACDFF
Entry 3: FFDDABAAABCDBBDDDDBBDDFFBCBBCC
Entry 4: CCCCFFBCBCCCCDFFFFBBBBDDBBCDAA
Entry 5: FFBBAACDDDFFAAFFBCCDBCAABBBCBB
```

```
Successfully processed 20 valid grade entries

Results:
Number of valid grade entries: 20
GLCS Length: 5
GLCS: BBCAB
```

```
Successfully processed 10 valid grade entries

Results:
Number of valid grade entries: 10
GLCS Length: 7
GLCS: CDDBBCC

Valid Grades Processed:
Entry 1: CDAAABBBCDBCBBBBAAABCDDDABCCAA
Entry 2: FFABABDDBCDDBBBCBCABABBCBBAAAA
Entry 3: AABCCCFFCCBBBCDDBCDDAABCCDCDAB
Entry 4: ABAABCABFFFFBCABBBBBDDBBFFABCC
Entry 5: CDCDCDAACCBBBCAABCFFCDCCDDABFF
Entry 6: BBBCFFCDDDBCBBCCAABCAACCFFDDDD
Entry 7: DDBCCDFFBCCCCCCCFFAAAACDCCBBCC
Entry 8: BCAABBDDBCAACDBBDDDDDDBCFFDDCC
Entry 9: BBCDDDBCAABBCCFFDDBBCDDDFFAABB
Entry 10: DDCCDDCDAAABCCCDBCDDBCFFBCABBB
```

```
Warning: No valid grades found in the file
No valid grade data found.
```

CONCLUSION:

In **Experiment Task 1**, we focused on finding the **Longest Common Subsequence (LCS)** of grades received by 20 students. The task was tackled by generating random sequences of grades and then applying dynamic programming techniques to identify the longest sequence of grades that appeared across all students' data. The program successfully handled valid grade entries and also managed edge cases such as missing or invalid grades. The LCS computation provided valuable insights into the common patterns in the grades of students, which could be useful in understanding general trends in performance. Overall, this task demonstrated the effectiveness of dynamic programming in solving sequence matching problems in real-world data.

Experiment Task 2

AIM: 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:

The Matrix Chain Multiplication (MCM) problem is an optimization technique that determines the most efficient order to multiply a series of matrices. Each city's meteorological data for a week is represented as a matrix where rows could correspond to daily readings of various parameters—temperature, dew point, wind speed, etc. while columns represent the different time intervals. Using MCM allows us to minimize the number of scalar multiplications required, making it possible to process this large data efficiently for timely weather predictions.

In meteorology, efficient computation is critical as weather data must be processed quickly for accurate forecasts. MCM helps by reducing the computational cost when handling numerous matrices, ensuring that predictions are made within a feasible timeframe. This optimization is especially valuable as meteorological data constantly updates, requiring frequent re-calculations.

To solve MCM, we employ a Dynamic Programming approach. We define a two-dimensional table where each cell represents the minimum multiplication cost for a sub-sequence of matrices. By systematically evaluating different partition points within the matrix chain, we find the optimal order of multiplication that minimizes the total operations. This solution ensures that even as data size grows, computational efficiency is maintained, supporting rapid and accurate weather predictions that are essential for responding to changing conditions.

PROGRAM:

```
#include <iostream>
#include <vector>
#include <limits>
#include <stdexcept>
#include <string>
using namespace std;

// Base class for matrix chain calculations
class MatrixChainBase {
protected:
    vector<pair<int, int>> dimensions_;
    int n_;

    MatrixChainBase(const vector<pair<int, int>>& dimensions)
        : dimensions_(dimensions), n_(dimensions.size()) {
        if (n_ == 0) throw runtime_error("No matrices available.");
    }

    void validateDimensions() const {
        for (size_t i = 0; i < dimensions_.size() - 1; ++i) {
            if (dimensions_[i].second != dimensions_[i + 1].first) {
                throw runtime_error("Matrix dimensions are not
conformable.");
            }
            if (dimensions_[i].first == 0 || dimensions_[i].second
== 0) {
                throw runtime_error("Zero dimensions are not
```

```
allowed.");
            }
        }
    }

    void printMatrixSequence(int i, int j) const {
        if (i == j) {
            cout << "C" << i + 1;
            return;
        }
        cout << "(";
        for (int k = i; k < j; ++k) {
            printMatrixSequence(i, k);
            cout << " x ";
            printMatrixSequence(k + 1, j);
        }
        cout << ")";
    }
};

// Optimized matrix chain multiplication with memoization
class MatrixChain : public MatrixChainBase {
private:
    vector<vector<int>> memo_;
    vector<vector<int>> sequence_;

    int calculateCost(int i, int j) {
        if (i == j) return 0;
        if (memo_[i][j] != -1) return memo_[i][j];

        int minCost = numeric_limits<int>::max();
        for (int k = i; k < j; ++k) {
            int cost = calculateCost(i, k) + calculateCost(k + 1, j)
+
                       dimensions_[i].first * dimensions_[k].second *
dimensions_[j].second;
            if (cost < minCost) {
                minCost = cost;
                sequence_[i][j] = k;
            }
        }
        memo_[i][j] = minCost;
        return minCost;
    }

    void printOptimalSequence(int i, int j) const {
```

```
        if (i == j) {
            cout << "C" << i + 1;
            return;
        }
        cout << "(";
        printOptimalSequence(i, sequence_[i][j]);
        printOptimalSequence(sequence_[i][j] + 1, j);
        cout << ")";
    }

public:
    MatrixChain(const vector<pair<int, int>>& dimensions)
        : MatrixChainBase(dimensions) {
        memo_ = vector<vector<int>>(n_, vector<int>(n_, -1));
        sequence_ = vector<vector<int>>(n_, vector<int>(n_, -1));
    }

    void displayResults() {
        try {
            validateDimensions();
            cout << "Original multiplication sequence cost: " <<
bruteForceMinCost() << endl;
            cout << "Optimal multiplication sequence cost: " <<
findOptimalCost() << endl;
            cout << "Original multiplication sequence: ";
            printMatrixSequence(0, n_ - 1);
            cout << "\nOptimal multiplication sequence: ";
            printOptimalSequence(0, n_ - 1);
            cout << endl << endl;
        }
        catch (const exception& e) {
            cout << e.what() << endl;
        }
    }

    int findOptimalCost() {
        return calculateCost(0, n_ - 1);
    }

    int bruteForceMinCost() const {
        int totalCost = 0;
        for (size_t i = 0; i < dimensions_.size() - 1; ++i) {
            totalCost += dimensions_[i].first *
dimensions_[i].second * dimensions_[i + 1].second;
        }
        return totalCost;
```

```
    }
};

// Brute force implementation
class MatrixChainBruteForce : public MatrixChainBase {
private:
    int bruteForceCost(int i, int j) {
        if (i == j) return 0;
        int minCost = numeric_limits<int>::max();
        for (int k = i; k < j; ++k) {
            int cost = bruteForceCost(i, k) + bruteForceCost(k + 1,
j) +
                       dimensions_[i].first * dimensions_[k].second *
dimensions_[j].second;
            minCost = min(minCost, cost);
        }
        return minCost;
    }

public:
    MatrixChainBruteForce(const vector<pair<int, int>>& dimensions)
        : MatrixChainBase(dimensions) {}

    void displayResults() {
        try {
            validateDimensions();
            cout << "Brute force minimum cost: " <<
bruteForceMinCost() << endl;
            cout << "Multiplication sequence: ";
            printMatrixSequence(0, n_ - 1);
            cout << endl;
        }
        catch (const exception& e) {
            cout << e.what() << endl;
        }
    }

    int bruteForceMinCost() {
        return bruteForceCost(0, n_ - 1);
    }
};

// Unified test class for both implementations
class MatrixChainTester {
private:
    void runSingleTest(const vector<pair<int, int>>& dimensions,
```

```
const string& testName) {
        cout << "\nRunning " << testName << endl;
        try {
            MatrixChain optimal(dimensions);
            MatrixChainBruteForce bruteForce(dimensions);

            optimal.displayResults();
            bruteForce.displayResults();
        }
        catch (const runtime_error& e) {
            cout << "Test error: " << e.what() << endl;
        }
    }

public:
    void runAllTests() {
        // Positive test cases
        cout << "Running Positive Test Cases:\n";
        vector<vector<pair<int, int>>> positiveTests = {
            {{6, 7}, {7, 5}, {5, 4}},
            {{6, 8}, {8, 5}, {5, 4}, {4, 6}},
            {{6, 8}, {8, 5}, {5, 4}, {4, 6}, {6, 3}},
            {{6, 8}, {8, 5}, {5, 4}, {4, 6}, {6, 3}, {3, 5}},
            {{6, 8}, {8, 5}, {5, 4}, {4, 6}, {6, 3}, {3, 5}, {5, 7},
{7, 2}}
        };

        for (size_t i = 0; i < positiveTests.size(); ++i) {
            runSingleTest(positiveTests[i], "Positive Test Case " +
to_string(i + 1));
        }

        // Negative test cases
        cout << "\nRunning Negative Test Cases:\n";
        vector<vector<pair<int, int>>> negativeTests = {
            {}, // Empty
            {{6, 7}, {5, 4}}, // Non-conformable
            {{6, 7}, {7, 0}, {0, 4}}, // Zero dimensions
            {{4, 4}, {4, 4}, {4, 4}} // Square matrices
        };

        for (size_t i = 0; i < negativeTests.size(); ++i) {
            runSingleTest(negativeTests[i], "Negative Test Case " +
to_string(i + 1));
        }
    }
```

```
};

int main() {
    vector<pair<int, int>> testDimensions = {{6, 7}, {7, 5}, {5,
4}};
    MatrixChainTester tester;
    tester.runAllTests();
    return 0;
}
```

OUTPUT:

```
Running Positive Test Cases:

 Running Test Case 1
Original multiplication sequence cost: 350
Optimal multiplication sequence cost: 308
Original multiplication sequence: C1 x C2 x C3
Optimal multiplication sequence: (C1(C2C3))


 Running Test Case 2
Original multiplication sequence cost: 520
Optimal multiplication sequence cost: 496
Original multiplication sequence: C1 x C2 x C3 x C4
Optimal multiplication sequence: ((C1(C2C3))C4)
```

```
 Running Test Case 3
Original multiplication sequence cost: 592
Optimal multiplication sequence cost: 396
Original multiplication sequence: C1 x C2 x C3 x C4 x C5
Optimal multiplication sequence: (C1(C2(C3(C4C5))))


 Running Test Case 4
Original multiplication sequence cost: 682
Optimal multiplication sequence cost: 486
Original multiplication sequence: C1 x C2 x C3 x C4 x C5
x C6
Optimal multiplication sequence: ((C1(C2(C3(C4C5))))C6)
```

```
 Running Test Case 5
Original multiplication sequence cost: 857
Optimal multiplication sequence cost: 400
Original multiplication sequence: C1 x C2 x C3 x C4 x C5
x C6 x C7 x C8
Optimal multiplication sequence: (C1(C2(C3(C4(C5(C6
(C7C8)))))))
```

```
 Running Negative Test Cases:

  Running Test Case 6
 No matrices available in  Test Case 6

  Running Test Case 7
  No multiplication possible for because mismatch in
  dimensions Test Case 7
 Original multiplication sequence cost: 0
 Optimal multiplication sequence cost: 0
 Original multiplication sequence: No sequence
 Optimal multiplication sequence: No sequence
```

```
 Running Test Case 8
No multiplication possible for because missing datas
highlighted by zero dimensions Test Case 8
Original multiplication sequence cost: 0
Optimal multiplication sequence cost: 0
Original multiplication sequence: No sequence
Optimal multiplication sequence: No sequence


 Running Test Case 9
Original multiplication sequence cost: 128
Optimal multiplication sequence cost: 128
Original multiplication sequence: C1 x C2 x C3
Optimal multiplication sequence: (C1(C2C3))
```

```
Running Positive Test Cases for Brute Force:

Running Test Case 1
Brute-force minimum multiplication sequence cost: 308
Original multiplication sequence cost: 350
Original multiplication sequence: C1 X C2 X C3
Brute-force multiplication sequence: (C1(C2C3))

Running Test Case 2
Brute-force minimum multiplication sequence cost: 496
Original multiplication sequence cost: 520
Original multiplication sequence: C1 X C2 X C3 X C4
Brute-force multiplication sequence: ((C1(C2C3))C4)
```

```
Running Test Case 3
Brute-force minimum multiplication sequence cost: 396
Original multiplication sequence cost: 592
Original multiplication sequence: C1 X C2 X C3 X C4 X C5
Brute-force multiplication sequence: (C1(C2(C3(C4C5))))

Running Test Case 4
Brute-force minimum multiplication sequence cost: 486
Original multiplication sequence cost: 682
Original multiplication sequence: C1 X C2 X C3 X C4 X C5
X C6
Brute-force multiplication sequence: (C1(C2(C3((C4C5))))
C6)
```

```
Running Test Case 5
Brute-force minimum multiplication sequence cost: 440
Original multiplication sequence cost: 857
Original multiplication sequence: C1 X C2 X C3 X C4 X C5
X C6 X C7 X C8
Brute-force multiplication sequence: (C1(C2(C3(C4(C5(C6
(C7C8)))))))
```

```
Running Negative Test Cases for Brute Force:

Running Test Case 6
No matrices available in  Test Case 6

Running Test Case 7
No multiplication possible for because mismatch in
dimensions Test Case 7
Brute-force minimum multiplication sequence cost: 0
Original multiplication sequence cost: 0
Original multiplication sequence: No sequence
Brute-force multiplication sequence: No sequence
```

```
Running Test Case 8
No multiplication possible for because of missing data
highlighted by zero dimensions Test Case 8
Brute-force minimum multiplication sequence cost: 0
Original multiplication sequence cost: 0
Original multiplication sequence: No sequence
Brute-force multiplication sequence: No sequence

Running Test Case 9
Brute-force minimum multiplication sequence cost: 128
Original multiplication sequence cost: 128
Original multiplication sequence: C1 X C2 X C3
Brute-force multiplication sequence: (C1(C2C3))
```

CONCLUSION:

In **Experiment Task 2**, the objective was to implement the **Matrix Chain Multiplication** algorithm to optimize the multiplication of matrices representing meteorological data, such as temperature, wind speed, and cloud cover, for multiple cities over a week. By applying the matrix chain multiplication algorithm, we were able to determine the most efficient sequence for performing matrix multiplications, ensuring quick and accurate predictions of weather patterns. The task highlighted the importance of optimization in handling large datasets, particularly in meteorological applications where timely predictions are critical. This experiment reinforced the relevance of matrix operations in real-world scenarios like weather forecasting.

THE END