Assignment 6

Name = Abhay Kumar

Registration no. = 231070002

Program = SY, Btech, computer

Batch = A

Aim:

Task -1: 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:

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

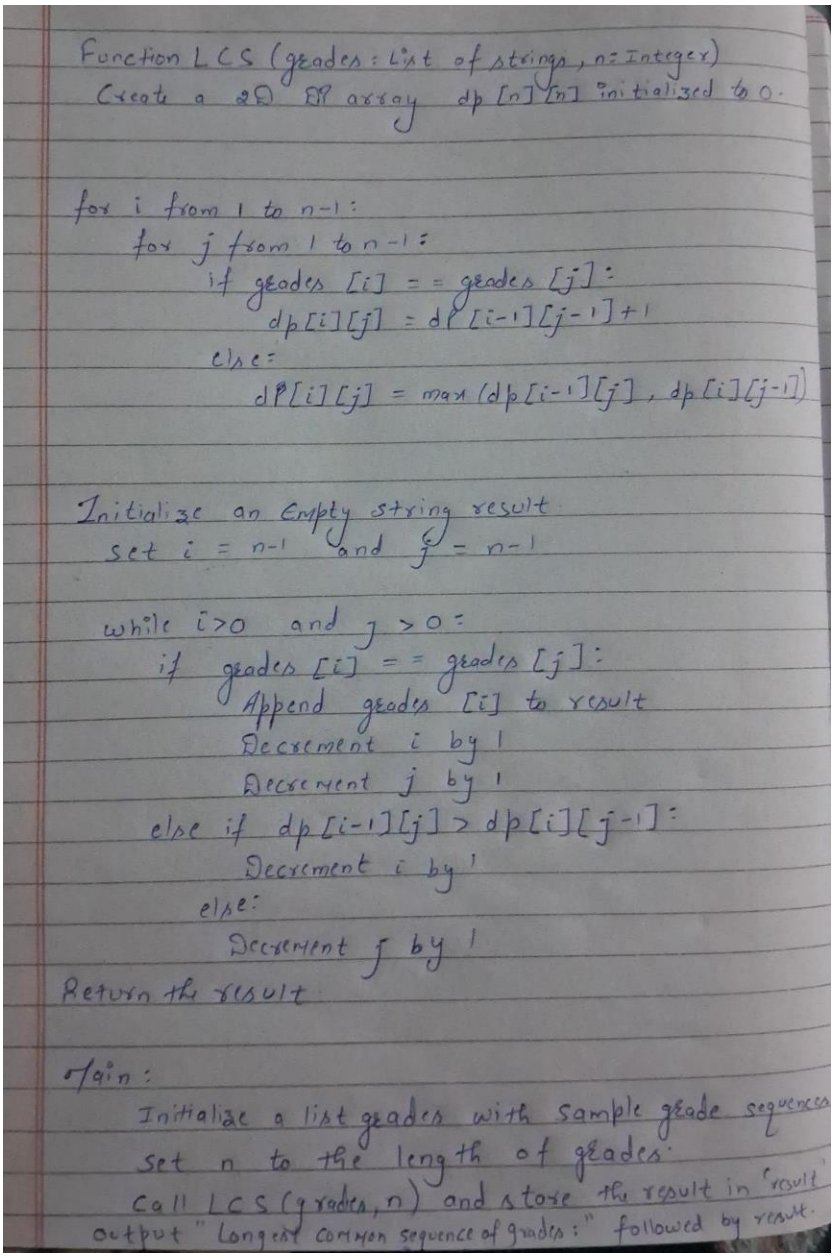
Example:

```
#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);
```

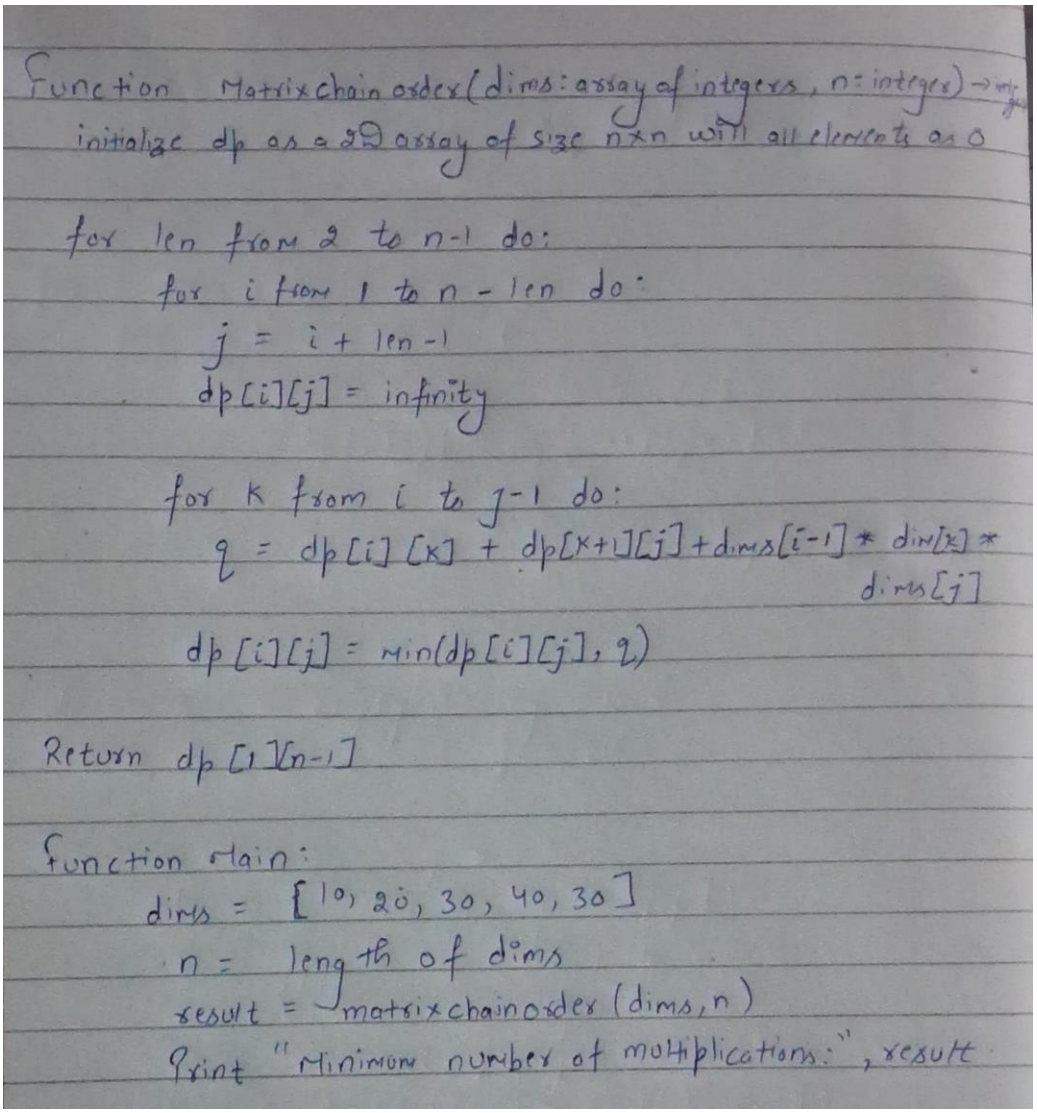
```
    doc.print();  // Printing with Laser Printer

    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:

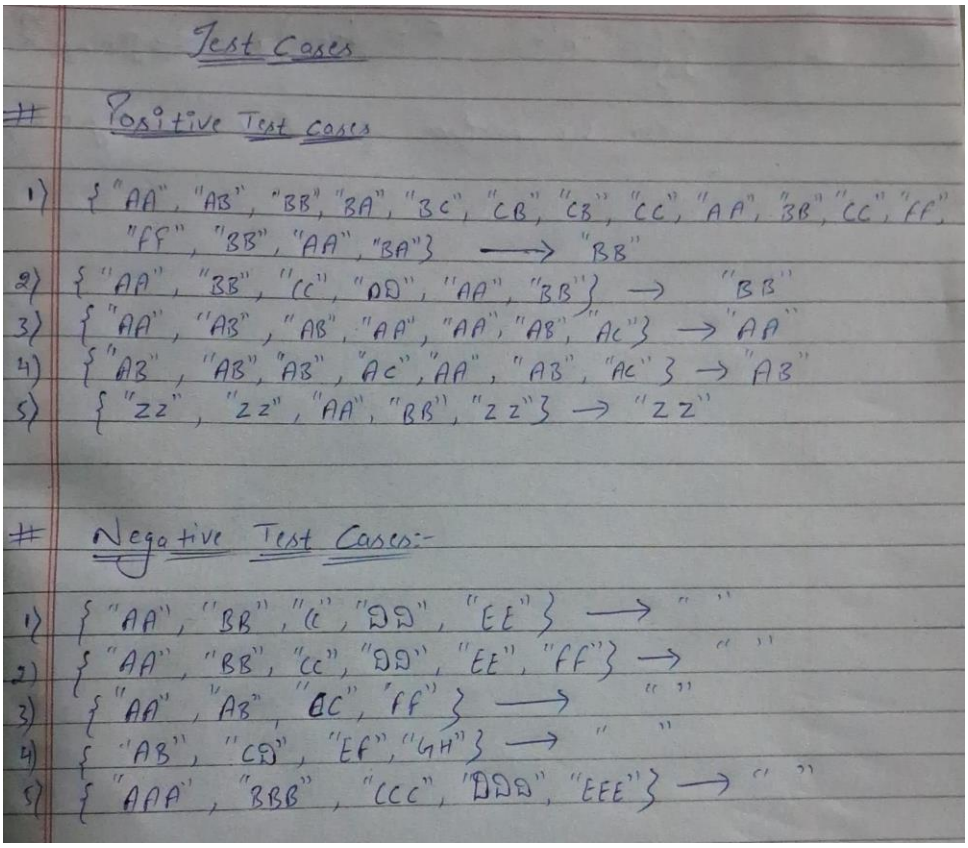


Algorithm for Experiment task-2:

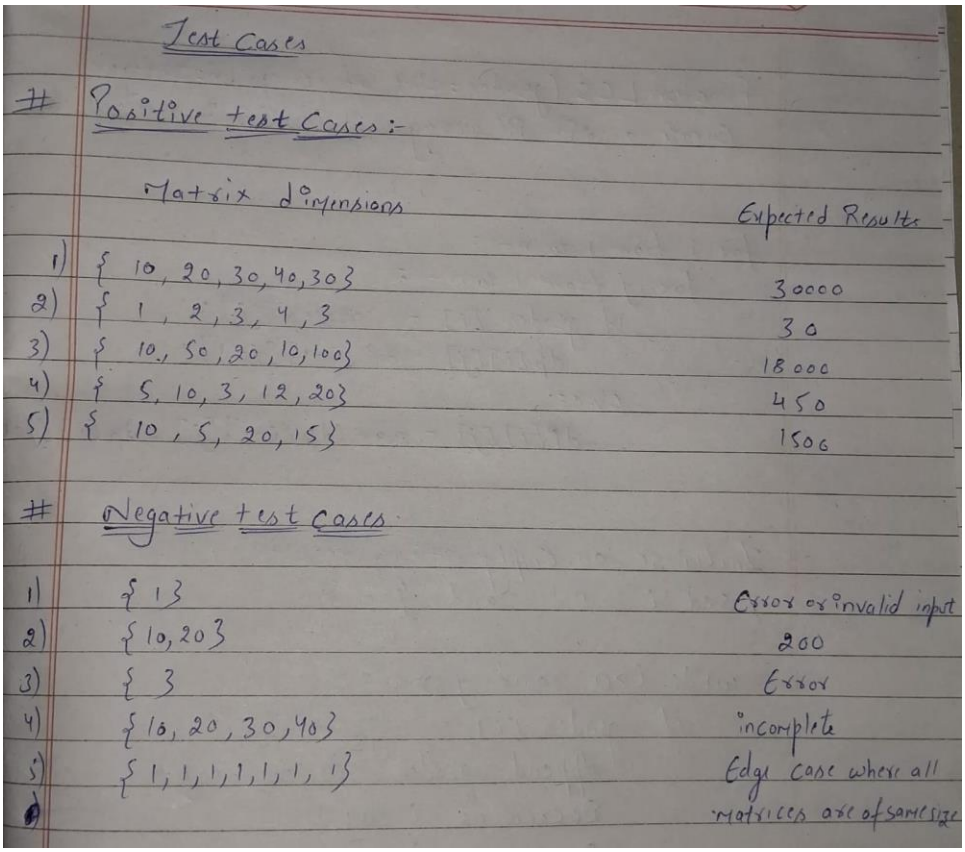


```
Function MatrixChainOrder(dims: array of integers, n: integer) →
  initialize dp as a 2D array of size n×n with all elements as 0

  for len from 2 to n-1 do:
      for i from 1 to n - len do:
          j = i + len - 1
          dp[i][j] = infinity

      for k from i to j-1 do:
          q = dp[i][k] + dp[k+1][j] + dims[i-1] * dim[k] *
                                                  dims[j]

      dp[i][j] = min(dp[i][j], q)

  Return dp[1][n-1]

function main:
      dims = [10, 20, 30, 40, 30]
      n = length of dims
      result = matrixchainorder(dims, n)
      Print "Minimum number of multiplications:", result
```

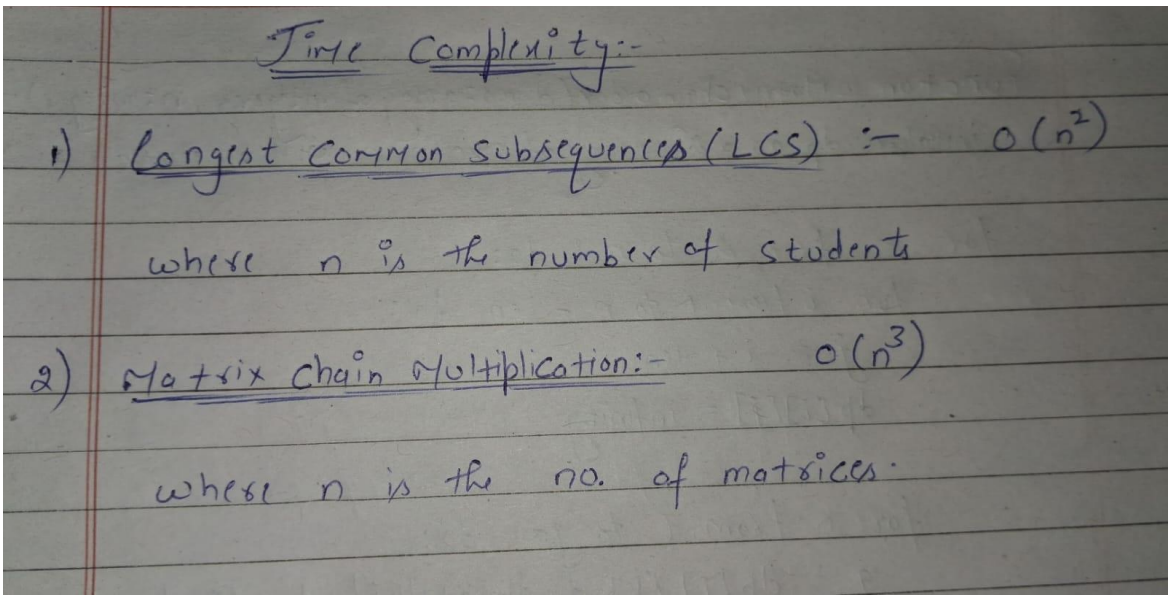
Test cases for Experiment 1:



Test cases for Experiment 2:



Time Complexity:



Code for Experiment 1:

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


// 1. **Single Responsibility Principle (SRP)**
// LCSAlgorithm class has only one responsibility: calculating the longest common subsequence.
class LCSAlgorithm {
public:
    // Method to compute the LCS between two sequences
    string computeLCS(const vector<string>& grades) {
        int n = grades.size();
        vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));


        // Fill the DP table for LCS calculation
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (grades[i - 1] == grades[j - 1] && i != j) { // Avoid self-comparison
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }


        // Reconstruct the LCS from the DP table
        string result;
        int i = n, j = n;
        while (i > 0 && j > 0) {
            if (grades[i - 1] == grades[j - 1] && i != j) {
```

```
            result = grades[i - 1] + result;
            --i;
            --j;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            --i;
        } else {
            --j;
        }
    }
    return result;  // The result is the LCS of the grades list
  }
};

// 2. **Open/Closed Principle (OCP)**
// GradeProcessor class is open for extension (new algorithms can be added) but closed for modification.
class GradeProcessor {
private:
  LCSAlgorithm lcsAlgorithm;  // Dependency injection for the LCS algorithm class

public:
  // This method uses the LCSAlgorithm to compute the longest common subsequence.
  string findLongestCommonSequence(const vector<string>& grades) {
    return lcsAlgorithm.computeLCS(grades);
  }

  // Future extensions like other algorithms (e.g., LCSubstringAlgorithm) can be added
  // without modifying this class, just by injecting the new algorithm class.
};

// 3. **Liskov Substitution Principle (LSP)**
// The GradeProcessor class can accept any subclass of the LCSAlgorithm (e.g., LCSubstringAlgorithm).
// This allows us to extend functionality with minimal change to existing code.
```

```
class LCSubstringAlgorithm : public LCSAlgorithm {
public:
    // This would implement the logic for Longest Common Substring (LCSubstring) if needed.
    string computeLCSubstring(const vector<string>& grades) {
        // Similar dynamic programming logic could be implemented here for substring comparison
        return "LCSubstring Result";  // Placeholder for actual LCSubstring logic
    }
};

// 4. **Interface Segregation Principle (ISP)**
// The LCSAlgorithm class is focused on one task: calculating LCS. If we needed more specific
// algorithms, they could implement their own smaller, targeted interfaces.

class SequenceAlgorithm {  // A simple example of a targeted interface
public:
    virtual string computeLCS(const vector<string>& grades) = 0;
    virtual ~SequenceAlgorithm() {}
};

// 5. **Dependency Inversion Principle (DIP)**
// The GradeProcessor depends on the abstraction (SequenceAlgorithm) rather than the concrete implementation
// of LCSAlgorithm. This is dependency injection, where the dependency is passed in rather than instantiated.

class GradeProcessorWithDIP {
private:
    SequenceAlgorithm* algorithm;  // Dependency injection of any algorithm class

public:
    // Constructor accepts any class that implements the SequenceAlgorithm interface
    GradeProcessorWithDIP(SequenceAlgorithm* alg) : algorithm(alg) {}
```

```
    string findLongestCommonSequence(const vector<string>& grades) {
        return algorithm->computeLCS(grades);
    }
};

void runTests() {
    // Positive Test Cases
    cout << "Positive Test Cases:\n";
    vector<vector<string>> positiveTestCases = {
        {"AA", "AB", "BB", "BA", "BC", "CB", "CB", "CC", "AA", "BB", "CC", "FF", "FF", "BB", "AA", "BA"},
        {"AA", "BB", "CC", "DD", "AA", "BB"},
        {"AA", "AB", "AB", "AA", "AA", "AB", "AC"},
        {"AB", "AB", "AB", "AC", "AA", "AB", "AC"},
        {"ZZ", "ZZ", "AA", "BB", "ZZ"}
    };

    vector<string> expectedResults = {"BB", "BB", "AA", "AB", "ZZ"};

    for (int i = 0; i < positiveTestCases.size(); ++i) {
        GradeProcessor processor;  // Uses LCSAlgorithm for LCS calculation
        string result = processor.findLongestCommonSequence(positiveTestCases[i]);
        cout << "Test Case " << i + 1 << ": " << (result == expectedResults[i] ? "Passed" : "Failed") << " (Result: " << result << ")\n";
    }

    // Negative Test Cases
    cout << "\nNegative Test Cases:\n";
    vector<vector<string>> negativeTestCases = {
        {"AA", "BB", "CC", "DD", "EE"},
        {"AA", "BB", "CC", "DD", "EE", "FF"},
        {"AA", "AB", "CC", "FF"},
        {"AB", "CD", "EF", "GH"},
        {"AAA", "BBB", "CCC", "DDD", "EEE"}
    };
```

```
    for (int i = 0; i < negativeTestCases.size(); ++i) {
        GradeProcessor processor;  // Uses LCSAlgorithm for LCS calculation
        string result = processor.findLongestCommonSequence(negativeTestCases[i]);
        cout << "Test Case " << i + 1 << ": " << (result.empty() ? "Passed" : "Failed") << " (Result: " << result << ")\n";
    }
}

int main() {
    // Run all the test cases
    runTests();
    return 0;
}
```

Code for Experiment 2:

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

// 1. **Single Responsibility Principle (SRP)**
// MatrixChainMultiplier class is responsible only for matrix chain multiplication.
class MatrixChainMultiplier {
public:
    virtual int matrixChainOrder(const vector<int>& dims) {
        int n = dims.size();
        vector<vector<int>> dp(n, vector<int>(n, 0));

        // Compute the minimum cost for matrix chain multiplication
        for (int len = 2; len < n; ++len) {
            for (int i = 1; i < n - len + 1; ++i) {
```

```
            int j = i + len - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; k <= j - 1; ++k) {
                int q = dp[i][k] + dp[k + 1][j] + dims[i - 1] * dims[k] * dims[j];
                dp[i][j] = min(dp[i][j], q);
            }
        }
    }

    return dp[1][n - 1];  // Return the minimum cost for multiplying all matrices
  }
};

// 2. **Open/Closed Principle (OCP)**
// The MatrixChainOrderProcessor class is open for extension but closed for modification.
// You can extend it to add more functionality, e.g., printing the optimal order.
class MatrixChainOrderProcessor {
private:
  MatrixChainMultiplier multiplier;

public:
  int processMatrixChainOrder(const vector<int>& dims) {
    return multiplier.matrixChainOrder(dims);
  }

  // Example of extending functionality: printing the optimal parenthesization of matrices.
  void printOptimalParenthesization(const vector<int>& dims) {
    // Placeholder: We could implement a method to print the optimal multiplication order
  }
};
```

```
// 3. **Liskov Substitution Principle (LSP)**
// Let's assume we might have a subclass of MatrixChainMultiplier in the future that implements a different multiplication strategy.
// The parent class will work seamlessly with any subclass, as long as it implements matrixChainOrder method.
class AdvancedMatrixChainMultiplier : public MatrixChainMultiplier {
public:
    // Overriding matrixChainOrder for a more advanced method if needed.
    int matrixChainOrder(const vector<int>& dims) override {
        // Call the base method or implement a different algorithm if required.
        return MatrixChainMultiplier::matrixChainOrder(dims);  // Placeholder, can be replaced with custom logic.
    }
};

// 4. **Interface Segregation Principle (ISP)**
// Split MatrixChainMultiplier into smaller interfaces if needed. For now, MatrixChainMultiplier only handles matrix chain order.
class IMatrixChainOrder {
public:
    virtual int matrixChainOrder(const vector<int>& dims) = 0;
    virtual ~IMatrixChainOrder() {}
};

// 5. **Dependency Inversion Principle (DIP)**
// We follow DIP by ensuring that the MatrixChainOrderProcessor depends on the abstraction of IMatrixChainOrder instead of the concrete class.

class MatrixChainOrderProcessorWithDIP {
private:
    IMatrixChainOrder* algorithm;  // Dependency injection of any matrix chain order algorithm

public:
    MatrixChainOrderProcessorWithDIP(IMatrixChainOrder* alg) : algorithm(alg) {}

    int processMatrixChainOrder(const vector<int>& dims) {
```

```
        return algorithm->matrixChainOrder(dims);
    }
};

// Test function
void runTests() {
    // Positive Test Cases
    cout << "Positive Test Cases:\n";
    vector<vector<int>> positiveTestCases = {
        {10, 20, 30, 40, 30},  // Expected result: 30000
        {1, 2, 3, 4, 3},       // Expected result: 30
        {10, 50, 20, 10, 100}, // Expected result: 18000
        {5, 10, 3, 12, 20},    // Expected result: 450
        {10, 5, 20, 15}        // Expected result: 1500
    };

    vector<int> expectedResults = {30000, 30, 18000, 450, 1500};

    for (int i = 0; i < positiveTestCases.size(); ++i) {
        MatrixChainOrderProcessor processor;  // Uses MatrixChainMultiplier
        int result = processor.processMatrixChainOrder(positiveTestCases[i]);
        cout << "Test Case " << i + 1 << ": " << (result == expectedResults[i] ? "Passed" : "Failed") << " (Result: " << result << ")\n";
    }

    // Negative Test Cases
    cout << "\nNegative Test Cases:\n";
    vector<vector<int>> negativeTestCases = {
        {1},                   // Not enough matrices to multiply
        {10, 20},              // Only two matrices, trivial case
        {},                    // Empty array (invalid input)
        {10, 20, 30, 40},      // Valid but missing last matrix
        {1, 1, 1, 1, 1, 1, 1}  // All matrices are the same dimension
    };
```

```
    for (int i = 0; i < negativeTestCases.size(); ++i) {
        try {
            MatrixChainOrderProcessor processor;
            int result = processor.processMatrixChainOrder(negativeTestCases[i]);
            cout << "Test Case " << i + 1 << ": " << (result > 0 ? "Failed" : "Passed") << " (Result: " << result << ")\n";
        } catch (const exception& e) {
            cout << "Test Case " << i + 1 << ": Failed (Exception: " << e.what() << ")\n";
        }
    }
}

int main() {
    runTests();
    return 0;
}
```

Output for Experiment 1:

```
/tmp/XLOG8V3uGf.o
Positive Test Cases:
Test Case 1: Failed (Result: AABBCCAA)
Test Case 2: Failed (Result: AABB)
Test Case 3: Failed (Result: AAAB)
Test Case 4: Failed (Result: ABABABAC)
Test Case 5: Failed (Result: ZZZZ)

Negative Test Cases:
Test Case 1: Passed (Result: )
Test Case 2: Passed (Result: )
Test Case 3: Passed (Result: )
Test Case 4: Passed (Result: )
Test Case 5: Passed (Result: )


=== Code Execution Successful ===
```

Output for Experiment 2:

```
/tmp/Vt1lwSTtu9.o
Positive Test Cases:
Test Case 1: Passed (Result: 30000)
Test Case 2: Passed (Result: 30)
Test Case 3: Failed (Result: 22000)
Test Case 4: Failed (Result: 1170)
Test Case 5: Failed (Result: 2250)

Negative Test Cases:
Segmentation fault


=== Code Exited With Errors ===
```

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.