Experiment 06

Algorithm FindLCS (string S1, string S2)

// Computes the longest common subsequences between 2 strings
// Input: Two strings S1 & S2
// Output: LCS of the strings

let m = S1.length and n = S2.length

declare a table of m+1 rows and n+1 columns of empty strings.

```
for i = 1 to m:
    for j = 1 to n:
        if S1[i-1] == S2[j-1]:
            table[i][j] = table[i-1][j-1] + 1
        else:
            table[i][j] = larger of
                          table[i-1][j] and
                          table[i][j-1] based
                          on their length.

return table[m][n]
```

Time Complexity:

a] let 'c' be the constant time taken for the innermost operation in the loops.

b] The first loop runs from 1 to m and 2nd loop runs from 1 to n.

c) Time $T = \sum_{i=1}^{m} \sum_{j=1}^{n} c$

$\therefore T = mnc$

$\therefore T \approx O(mn)$

Hence, time complexity is $O(mn)$ where m & n are the lengths of the string.

★ Algorithm FindLCSMultiple (string S).

```
//Compute the LCS among n strings
//Input : Array of n strings
//Output : LCS of n strings

seq = " "
for i = 0 to n-1 :
        LCS = S[i]
        for j = 0 to n-1 :
            if i ≠ j :
                LCS = FindLCS(LCS, S[j])
                if LCS == " " :
                        break

            seq = max( seq, LCS,
                    key = length).

return seq
```

Time Complexity:

a] There are two loops, one from 0 to $n-1$ and from 0 to $n-1$.

b] In the innermost operation, computing the LCS is $O(L^2)$ in the worst case.

[Assume an average string length of L]

c] Hence the total time complexity is

$$T = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} cL^2 = O(n^2L^2)$$

Testcases:

a] ["AABBCCDD", "BBCCDAA"]
→ ~~CD~~ ~~B~~ BBCCD

b] ["BCBCCDFF", "FFBCAADD"]
→ BCD

c] ["ABABDDCD", "FFDDFFCD"]
→ DDCD

d] ["AAAAAAAA", "ABBBBBBB"]
→ A

e] ["FFFFFFFF", "FFFFFFFF"]
→ FFFFFFFF

f] ["", "AABBCCDDFF"]
→ no char in string

g] ["BBFFCCAADD", "AADDFF34"]
→ string should not contain numbers

h] ["AAAAAAAAAA", "BBBBBBBB"]
→ no common subsequence found

i] ["AABBDDFFAA", "AABBFF44DD"]
→ string should not contain numbers.

```
Positive Test Cases:
Test Case 1: BBCCD
Test Case 2: BCD
Test Case 3: DDCD
Test Case 4: A
Test Case 5: FFFFFFFF

Negative Test Cases:
Test Case 6: No characters in the string
Test Case 7: Grade sequences should not contain numbers
Test Case 8: No common subsequence found
Test Case 9: Grade sequences should not contain numbers
```

```
Algorithm MatrixChainMultiplication (N, arr):
// Input: An array containing the matrix
        dimensions.
// Output: smallest number of multiplication
        and optimal order.

define dp[1.....N][1.....N] as a 2D array to
store minimum multiplication costs.

define S[1.....N][1.....N] as the split points.

for L=2 to N-1:
    for i=1 to N-L:
        j = i+L-1
        dp[i][j] = ∞
        for k=i to j-1:
            q = dp[i][k] + dp[k+1][j]
                + arr[i-1]*arr[k]*arr[j]
        if q < dp[i][j]:
            dp[i][j] = q
            S[i][j] = k

optimal_order = Getoptimalorder(1, N-1)

return dp[1][N-1], optimal_order

getoptimalorder(i, j):
    if i==j return "M" + i
    k = split[i][j]
    left = getoptimalorder(i, k)
    right = getoptimalorder(k+1, j)
    return ({left} x {right})
```

Time Complexity:

a] There are 3 nested loops in the algorithm.

- Outer loop: $L = 2$ to $N-1$
- middle loop: $i = 1$ to $N-L$
- inner loop: $k = i$ to $i+L-2$

b] summing up the loop operations.

$$T = \sum_{L=2}^{N-1} \sum_{i=1}^{N-L} \sum_{k=i}^{i+L-2} O(1) \text{ [basic operation]}$$

$$= \sum_{L=2}^{N-1} \sum_{i=1}^{N-L} O(L)$$

$$= O\left(\sum_{L=2}^{N-2} (N-L)(L) \right)$$

$$= O\left(N \sum_{L=2}^{N-1} L - \sum_{L=2}^{N-2} L^2 \right)$$

$$\approx O\left(\frac{NN^2}{2} - \frac{N^3}{6} \right)$$

$$\approx O(N^3).$$

Testcases

1] [15, 25, 35, 45]
→ 36750

2] [12, 90, 15, 60, 3]
→ 9990

3] [8, 25, 10, 50, 30, 5]
→ 12250

4] [7, 8, 9]
→ 504

5] [10, 20]
→ 0

6] []
→ list cant be empty

7] ['-b', 25, 35, 45]
→ entries should be +ve numbers

8] [15, -30, 25, 50]
→ entries should be +ve numbers

9] [10]
→ list should be atleast 2 numbers

10] [100, 20, 30, -40, 30]
→ entries should be +ve numbers.

```
---- Positive Test Cases ----

Test Case 1 - 3 Matrices - Matrix Dimensions: [15, 25, 35, 45]
Minimum number of scalar multiplications: 36750

Test Case 2 - 4 Matrices - Matrix Dimensions: [12, 90, 15, 60, 3]
Minimum number of scalar multiplications: 9990

Test Case 3 - 5 Matrices - Matrix Dimensions: [8, 25, 10, 50, 30, 5]
Minimum number of scalar multiplications: 12250

Test Case 4 - 2 Matrices - Matrix Dimensions: [7, 8, 9]
Minimum number of scalar multiplications: 504

Test Case 5 - Single Matrix (Edge Case) - Matrix Dimensions: [10, 20]
Minimum number of scalar multiplications: 0

---- Negative Test Cases ----

Test Case 1 - Empty Matrix Dimensions - Matrix Dimensions: []
Error: Matrix dimensions list must contain at least two values representing matrix chains.

Test Case 2 - Non-Numeric Matrix Dimension - Matrix Dimensions: ['b', 25, 35, 45]
Error: Matrix dimensions must be positive integers.

Test Case 3 - Negative Matrix Dimension - Matrix Dimensions: [15, -30, 25, 50]
Error: Matrix dimensions must be positive integers.

Test Case 4 - Single Matrix (No Chain to Multiply) - Matrix Dimensions: [10]
Error: Matrix dimensions list must contain at least two values representing matrix chains.

Test Case 5 - Inconsistent Matrix Dimensions - Matrix Dimensions: [100, 20, 30, -40, 30]
Error: Matrix dimensions must be positive integers.
```

Conclusion:

a] We have implemented the longest common subsequence algorithm using dynamic programming and found the longest common subsequence for 20 sequences of grades.

b] We have found the least number of multiplications to multiply metreorological matrix data using dynamic programming.

c] We have studied and implemented the 5 SOLID Principles using sample classes and objects.

1. Single Responsibility Principle (SRP)

- **Definition**: A class should have only one reason to change, meaning it should only have one job or responsibility.
- **Example**: Consider a `Book` class. If this class has methods for printing book details, saving book data to a file, and performing text analysis, it violates SRP. Instead, you should separate these into different classes like `Book`, `BookPrinter`, and `BookSaver`.

```
# Violating SRP
class Book:
    def get_title(self):
        return "Title"

    def print_book(self):
        print("Printing book details...")

    def save_to_file(self):
        with open('book.txt', 'w') as file:
            file.write(self.get_title())
# Following SRP
class Book:
    def get_title(self):
        return "Title"

class BookPrinter:
    def print_book(self, book):
        print("Printing book details...")

class BookSaver:
    def save_to_file(self, book):
        with open('book.txt', 'w') as file:
            file.write(book.get_title())
```

2. Open/Closed Principle (OCP)

- **Definition**: Software entities (classes, modules, functions) should be open for extension but closed for modification.
- **Example**: Suppose you have a `Shape` class. Instead of modifying it every time you add a new shape, you should use polymorphism to extend it.

```
# Violating OCP
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Adding a new shape requires modifying existing code
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

```
    def area(self):
        return 3.14 * self.radius * self.radius
# Following OCP
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius
```

3. Liskov Substitution Principle (LSP)

- **Definition**: Subtypes should be substitutable for their base types without altering the correctness of the program.
- **Example**: If a subclass overrides a method of a parent class, it should not break the parent class's expectations.

```
# Violating LSP
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Flying...")

class Ostrich(Bird):
    def fly(self):
        raise Exception("Ostriches can't fly!")  # Violates LSP
# Following LSP
class Bird:
    pass

class FlyingBird(Bird):
    def fly(self):
        pass

class Sparrow(FlyingBird):
    def fly(self):
        print("Flying...")

class Ostrich(Bird):
    pass
```

4. Interface Segregation Principle (ISP)

- **Definition**: A client should not be forced to implement interfaces it does not use. Instead of one large interface, prefer smaller, more specific ones.
- **Example**: If you have a `Worker` interface that requires implementing `eat()` and `work()` methods, it would be inappropriate for a robot worker to have to implement `eat()`.

```
# Violating ISP
class Worker:
    def eat(self):
        pass

    def work(self):
        pass

class HumanWorker(Worker):
    def eat(self):
        print("Eating...")

    def work(self):
        print("Working...")

class RobotWorker(Worker):
    def eat(self):
        raise Exception("Robots don't eat")  # Violates ISP

    def work(self):
        print("Working...")
# Following ISP
class Workable:
    def work(self):
        pass

class Eatable:
    def eat(self):
        pass

class HumanWorker(Workable, Eatable):
    def work(self):
        print("Working...")

    def eat(self):
        print("Eating...")

class RobotWorker(Workable):
    def work(self):
        print("Working...")
```

5. Dependency Inversion Principle (DIP)

- **Definition**: 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.
- **Example**: Instead of a class depending on a specific database implementation, depend on an interface.

```python
# Violating DIP
class SQLDatabase:
    def connect(self):
        print("Connecting to SQL database...")

class Application:
    def __init__(self):
        self.database = SQLDatabase()

    def start(self):
        self.database.connect()
# Following DIP
class Database(ABC):
    @abstractmethod
    def connect(self):
        pass

class SQLDatabase(Database):
    def connect(self):
        print("Connecting to SQL database...")

class Application:
    def __init__(self, database: Database):
        self.database = database

    def start(self):
        self.database.connect()

# Now, the Application class can work with any Database implementation.
db = SQLDatabase()
app = Application(db)
app.start()
```

By following the SOLID principles, your code will generally be more maintainable, extensible, and easier to understand.