Name: Ahaan Desai
Roll no.: 231070016
Batch: Batch A (1-20)

DAA Lab Experiment 6

Aim

1. To find longest common subsequence of sequence of grades received by 20 students
2. To find fastest way to multiply matrices of meteorological data using matrix chain multiplication algorithm
3. To understand and implement SOLID principles of software development

Program

1. Longest Common Subsequence

```python
import pandas

def lcs(seq1, seq2):
    """
    Computes the longest common subsequence (LCS) between one string.
    The dynamic programming table stores the actual LCS as the state, instead
of the length.
    """
    dp = [[""] * (len(seq2) + 1) for _ in range(len(seq1) + 1)]

    for i in range(1, len(seq1) + 1):
        for j in range(1, len(seq2) + 1):
            if seq1[i - 1] == seq2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + seq1[i - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], key=len)
                # key = len means the strings are compared on basis of length;
the string with larger length is used

    return dp[-1][-1]

def lcs_n_strings(sequences):
    """
    Computes the longest common subsequence among n strings.
    It takes a base string and compares it with all other strings.
    The largest LCS obtained from each base string is considered.
    """
    if len(sequences) == 0:
        return "No sequences provided", -1

    if not all(len(s) == len(sequences[0]) for s in sequences):
        return "All sequences must have the same length", -1

    max_sequence = ""
    for i in range(len(sequences)):
        common_sequence = sequences[i]
        for j in range(len(sequences)):
            if i == j: continue
            common_sequence = lcs(common_sequence, sequences[j])
            if not common_sequence:
                break
        max_sequence = max(max_sequence, common_sequence, key=len)
    return max_sequence, 1

def process_grades(student_grades):
    """
    Reads the student grades from a CSV file and computes the longest common
sequence among them.
    """
    for grades in student_grades:
        grades = [grade for grade in grades if pandas.notna(grade) and grade !=
""]
        if len(grades) == 0:
            print("Error: No grades provided")
            continue
        if len(grades) < 20:
            print("Error: Number of sequences less than 20")
            continue

        lcs_result, status = lcs_n_strings(grades)
        if status == 1:
            print(f"Longest common sequence: {lcs_result}. Length of sequence:
{len(lcs_result)}")
        else:
            print(f"Error: {lcs_result}")

if __name__ == '__main__':
       student_grades =
       pandas.read_csv('student_grades.csv').fillna("").to_numpy()
       process_grades(student_grades)
```

2. Matrix Chain Multiplication

Python
```python
def matrix_chain_multiplication(N, arr):
    """
    Computes the optimal order of multiplying matrices to minimize the number
of scalar multiplications.
    """
    if len(arr) < 2:
        return 0, "No matrices provided"
    if len(arr) == 2:
        return 0, "Only one matrix provided"
    if any(k < 0 for k in arr):
        return -1, "Matrix dimensions must be positive"

    dp = [[0 for _ in range(N)] for _ in range(N)]
    split = [[0 for _ in range(N)] for _ in range(N)]

    for L in range(2, N):
        for i in range(1, N - L + 1):
            j = i + L - 1
            dp[i][j] = float('inf')

            for k in range(i, j):
                q = dp[i][k] + dp[k + 1][j] + arr[i - 1] * arr[k] * arr[j]
                if q < dp[i][j]:
                    dp[i][j] = q
                    split[i][j] = k
    def get_optimal_order(i, j):
        if i == j:
            return f"A{i}"
        k = split[i][j]
        left_order = get_optimal_order(i, k)
        right_order = get_optimal_order(k + 1, j)
        return f"({left_order} x {right_order})"

    optimal_order = get_optimal_order(1, N - 1)
    return dp[1][N - 1], optimal_order

def tests():
    """
    Finds the optimal order of multiplying matrices for different
metereological matrix dimensions.
    """
    test_cases = [
        [7, 3, 7, 4, 7, 5, 7, 6],
        [7, 5, 7, 6, 7, 7, 7, 8],
        [7, 4, 7, 8, 7, 3, 7, 9],
        [7, 2, 7, 10, 7, 4, 7, 5],
        [7, 6, 7, 3, 7, 8, 7, 2],
        [7, 9, 7, 5, -7, 10, -7, 3],
        [7, 7],
        [7, -10, 7, 3, -7, 9, 7, 5],
        [10],
        []
    ]
    for tc in test_cases:
        print(matrix_chain_multiplication(len(tc), tc))

if __name__ == '__main__':
      tests()
```

3. SOLID Principles

1. Single Responsibility Principle

```python
"""
Single Responsibility Principle (SPR)
Definition: A class should have only one reason to change, meaning that a class
should have only one job or responsibility.
"""

class OrderProcessor:
    def process(self, order):
        print("Processing order:", order)

class OrderRepository:
    def save(self, order):
        print("Saving order to database:", order)


# Each class only has one responsibility - OrderProcessor processes orders and
OrderRepository saves orders to the database.

# Usage
order = {"id": 1, "items": ["apple", "banana"]}
processor = OrderProcessor()
repository = OrderRepository()
processor.process(order)
repository.save(order)
```

2. Open/Closed principle

```python
"""
Open/Closed Principle
Definition: A class should be open for extension but closed for modification.
"""

from abc import ABC, abstractmethod
# Abstract base class module
```

```
class Notification(ABC):
    """Defines an abstract class which can be used as a template for other
types of notifications."""
    @abstractmethod
    def send(self, message):
        # Do nothing
        pass

class EmailNotification(Notification):
    """Class for email notifications, inherits from notifications template."""
    def send(self, message):
        print("Sending email:", message)

class SMSNotification(Notification):
    """Class for SMS notifications, inheritsfrom notifications template."""
    def send(self, message):
        print("Sending SMS:", message)


# In here the notification class cannot be modified but can be extended to add
new types of notifications.

# Usage
notifications = [EmailNotification(), SMSNotification()]
for notifier in notifications:
    notifier.send("Hello, World!")
```

3. Liskov substitution principle

Python
```
"""
Liskov Substitution Principle
Definition: Objects of a superclass should be replaceable with objects of its
subclasses without affecting the functionality of the program.
"""
from abc import ABC, abstractmethod

class Bird(ABC):
    @abstractmethod
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flying")

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Penguins can't fly!")

def make_bird_fly(bird: Bird):
    try:
        bird.fly()
    except NotImplementedError as e:
        print(e)

# Usage
sparrow = Sparrow()
penguin = Penguin()
make_bird_fly(sparrow)
make_bird_fly(penguin)

# In here the function is defined for the 'Bird' type but objects of the
sparrow type can also be used in the same function.
# Objects of the penguin type cannot be used as on calling the 'fly' function
an error is thrown
```

4. Interface segregation principle

```
Python
"""
Interface Segregation Principle
Definition: A client should never be forced to implement an interface that it
does not use, or clients should not be forced to depend on methods they do not
use.
"""
from abc import ABC, abstractmethod

class Printer(ABC):
    @abstractmethod
    def print_document(self):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan_document(self):
        pass

class AllInOnePrinter(Printer, Scanner):
    def print_document(self):
        print("Printing document")

    def scan_document(self):
        print("Scanning document")

class BasicPrinter(Printer):
    def print_document(self):
        print("Printing document")

# Here the basic printer does not need to implement the scan_document method as
it is not required for its functionality

# Usage
aio_printer = AllInOnePrinter()
basic_printer = BasicPrinter()
aio_printer.print_document()
aio_printer.scan_document()
basic_printer.print_document()
```

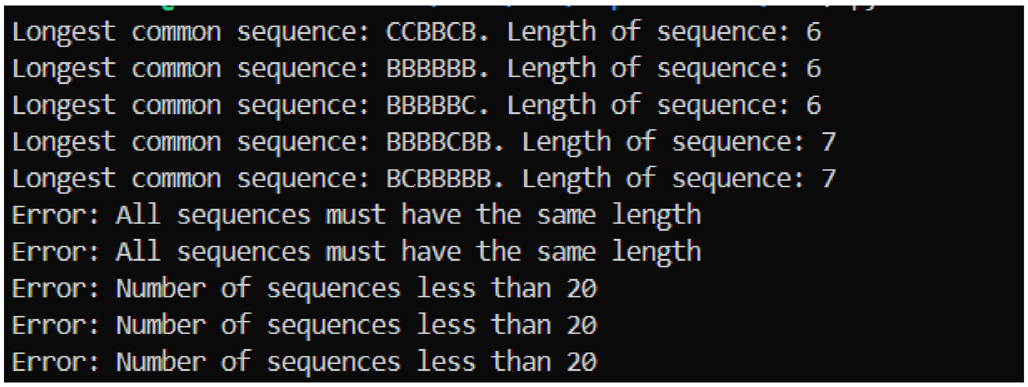
5. Dependency inversion principle

Python
```
"""
Dependency Inversion Principle (DIP)
Definition: High-level modules should not depend on low-level modules; both
should depend on abstractions.
"""
from abc import ABC, abstractmethod

class Database(ABC):
    @abstractmethod
    def save(self, data):
```

```python
        pass

class MySQLDatabase(Database):
    def save(self, data):
        print("Saving data to MySQL:", data)

class MongoDB(Database):
    def save(self, data):
        print("Saving data to MongoDB:", data)

class DataHandler:
    def __init__(self, db: Database):
        self.db = db

    def save_data(self, data):
        self.db.save(data)

# The mongodb and mysql handler only depend on the database interface, and not
on each other
# We should be able to switch between two databases without changing the
handler's code
# Usage
mysql_db = MySQLDatabase()
mongo_db = MongoDB()
handler = DataHandler(mysql_db)
handler.save_data("Order Data")
handler = DataHandler(mongo_db)
handler.save_data("User Data")
```

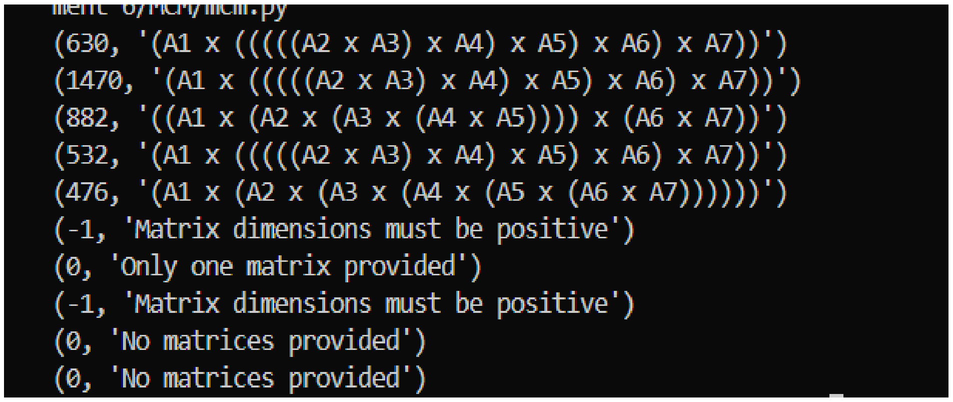
Output

1. Longest Common Subsequence



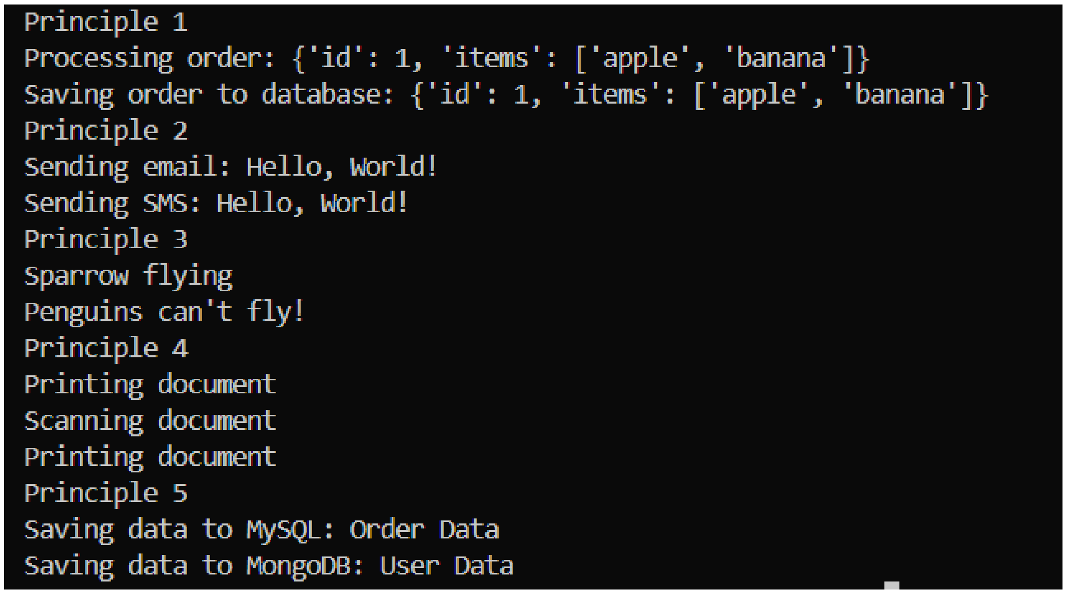
```
Longest common sequence: CCBBCB. Length of sequence: 6
Longest common sequence: BBBBBB. Length of sequence: 6
Longest common sequence: BBBBBC. Length of sequence: 6
Longest common sequence: BBBBCBB. Length of sequence: 7
Longest common sequence: BCBBBBB. Length of sequence: 7
Error: All sequences must have the same length
Error: All sequences must have the same length
Error: Number of sequences less than 20
Error: Number of sequences less than 20
Error: Number of sequences less than 20
```

2. Matrix Chain Multiplication



```
  (630, '(A1 x (((((A2 x A3) x A4) x A5) x A6) x A7))')
  (1470, '(A1 x (((((A2 x A3) x A4) x A5) x A6) x A7))')
  (882, '((A1 x (A2 x (A3 x (A4 x A5)))) x (A6 x A7))')
  (532, '(A1 x (((((A2 x A3) x A4) x A5) x A6) x A7))')
  (476, '(A1 x (A2 x (A3 x (A4 x (A5 x (A6 x A7))))))')
  (-1, 'Matrix dimensions must be positive')
  (0, 'Only one matrix provided')
  (-1, 'Matrix dimensions must be positive')
  (0, 'No matrices provided')
  (0, 'No matrices provided')
```

3. SOLID Principles



```
Principle 1
Processing order: {'id': 1, 'items': ['apple', 'banana']}
Saving order to database: {'id': 1, 'items': ['apple', 'banana']}
Principle 2
Sending email: Hello, World!
Sending SMS: Hello, World!
Principle 3
Sparrow flying
Penguins can't fly!
Principle 4
Printing document
Scanning document
Printing document
Principle 5
Saving data to MySQL: Order Data
Saving data to MongoDB: User Data
```

Conclusion

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

2. We have found the least number of multiplications to multiply meteorological matrix data using dynamic programming.

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