EXPERIMENT NO. 6

NAME: ADITYA B. KULKARNI
REGISTRATION NO : 241070908
SUBJECT : DAA (LAB)
SY BTECH COMPUTER ENGINEERING

Aim:

1. Task-1:- Consider grades received by 20 students, like AA, AB, BB, ..., FF of each student.
   Compute the Longest common sequence of grades among students.

2. 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 :

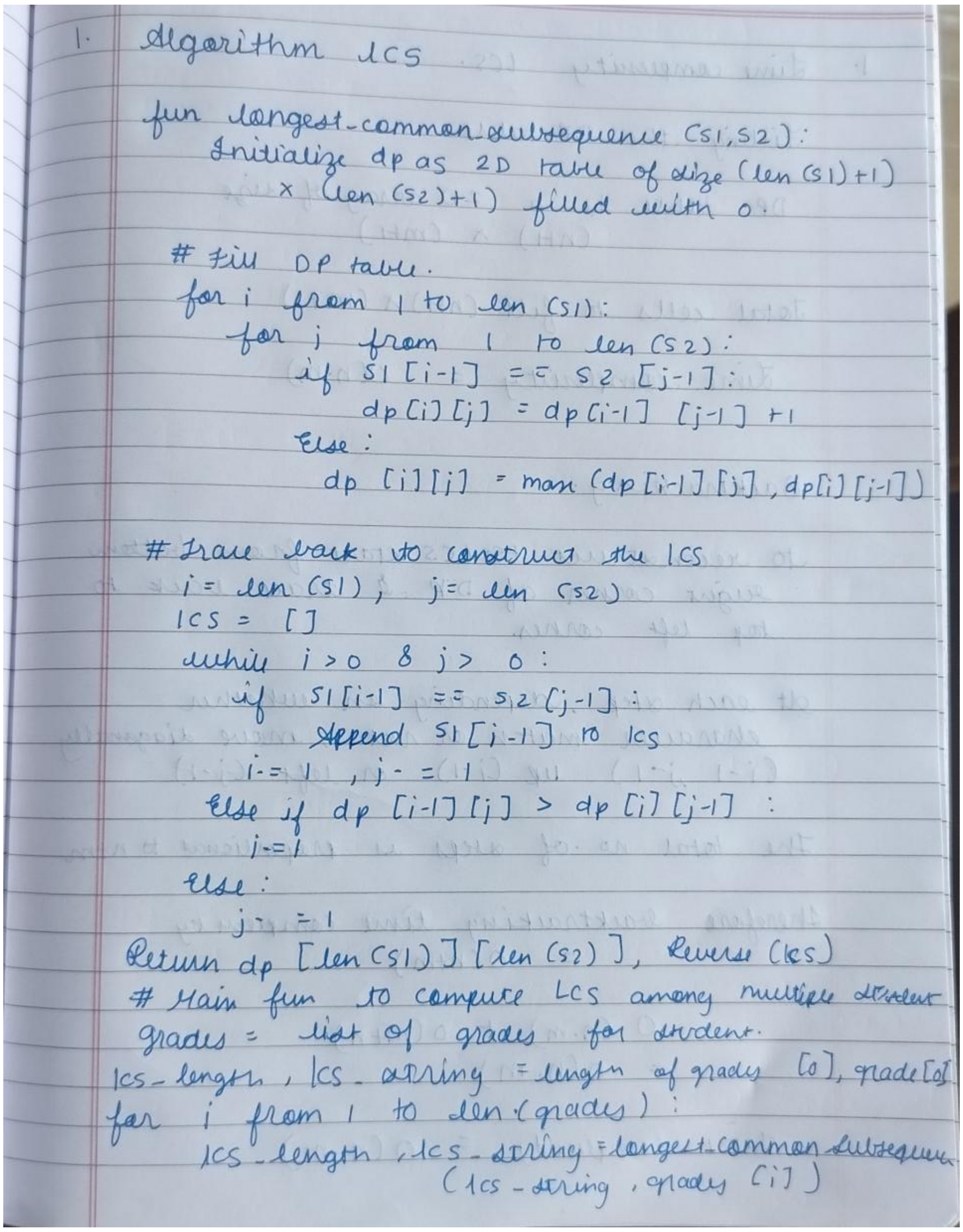
Longest Common Subsequence (LCS):is a problem that finds the longest sequence that can be derived from two sequences (often strings), where the elements of the sequence appear in the same order, but not necessarily consecutively.

- Dynamic Programming:is a problem-solving approach used to solve complex problems by breaking them down into simpler subproblems. It is particularly useful when the same subproblems are solved multiple times, as it stores the results of subproblems (in a table or array) to avoid redundant computations. This technique is also called memoization when storing results in a cache or tabulation when using bottom-up iteration.

Key Concepts in Dyanmic Programming:

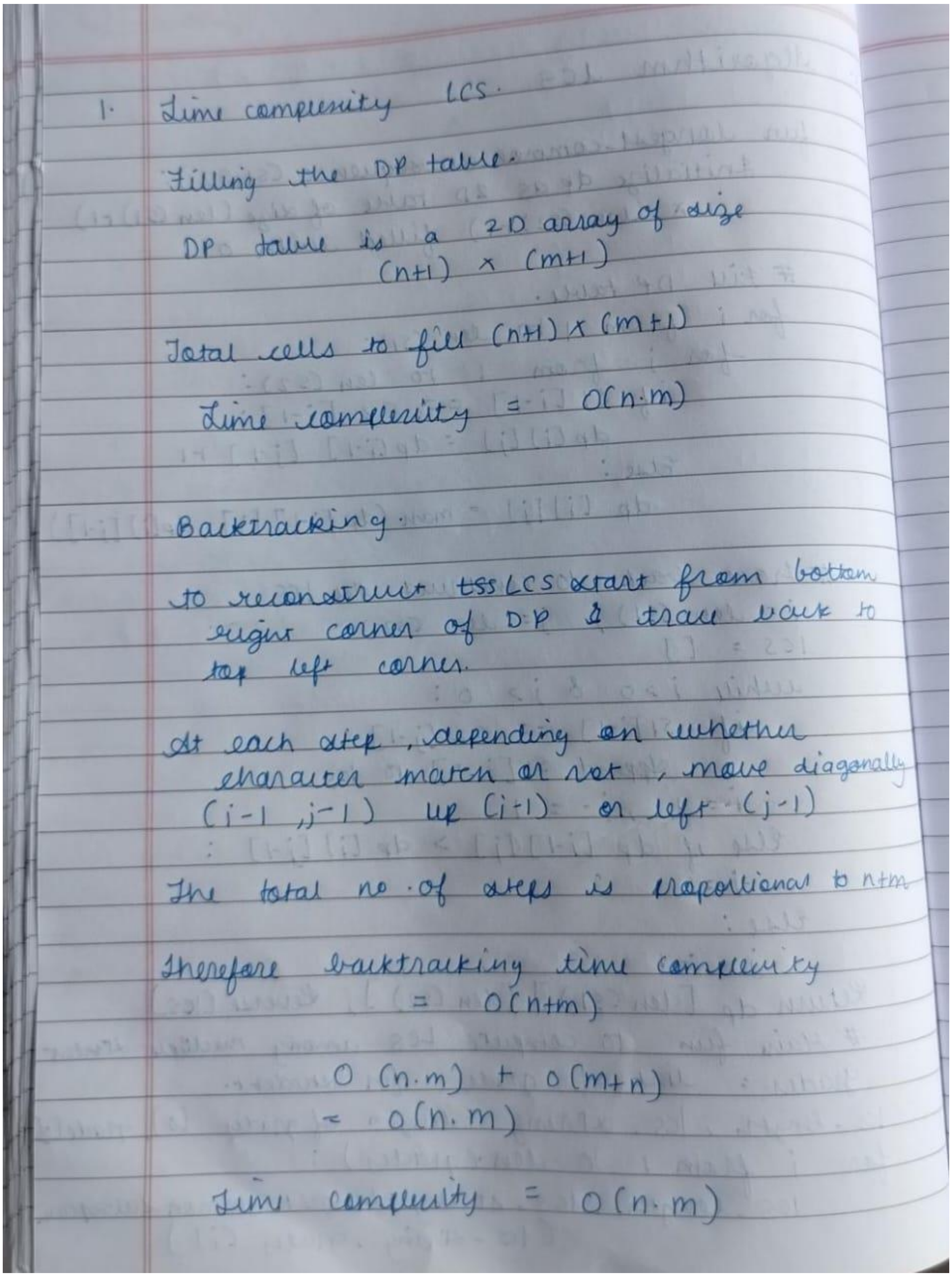
1. Optimal Substructure:
2. Overlapping Subproblems:
3. State:
4. Transition

Algorithm LCS:



```
1. Algorithm LCS

fun longest-common-subsequence (s1, s2):
    Initialize dp as 2D table of size (len (s1)+1)
        x (len (s2)+1) filled with 0.

  # Fill DP table.
  for i from 1 to len (s1):
      for j from 1 to len (s2):
          if s1 [i-1] == s2 [j-1]:
              dp [i] [j] = dp [i-1] [j-1] +1
          Else:
              dp [i][j] = max (dp [i-1] [j], dp[i] [j-1])

  # Trace back to construct the LCS
  i = len (s1) ;  j = len (s2)
  lcs = []
  while i > 0 & j > 0 :
      if s1 [i-1] == s2 [j-1] :
          Append s1 [i-1] to lcs
          i -= 1 , j -= 1
      Else if dp [i-1] [j] > dp [i] [j-1] :
          i -= 1
      Else :
          j -= 1
Return dp [len (s1)] [len (s2)], Reverse (lcs)
# Main fun to compute LCS among multiple student
grades = list of grades for student.
lcs-length, lcs-string = length of grades [0], grade[0]
for i from 1 to len (grades):
    lcs-length, lcs-string = longest-common-subsequen
                    (lcs-string, grades [i])
```

Time complexity LCS:



1. Time complexity LCS.

Filling the DP table.

DP table is a 2D array of size
(n+1) x (m+1)

Total cells to fill (n+1) x (m+1)

Time complexity = O(n·m)

Backtracking.

To reconstruct ~~tss~~ LCS start from bottom right corner of DP & trace back to top left corner.

At each step, depending on whether character match or not, move diagonally (i-1, j-1) up (i-1) or left (j-1)

The total no. of steps is proportional to n+m

Therefore backtracking time complexity
= O(n+m)

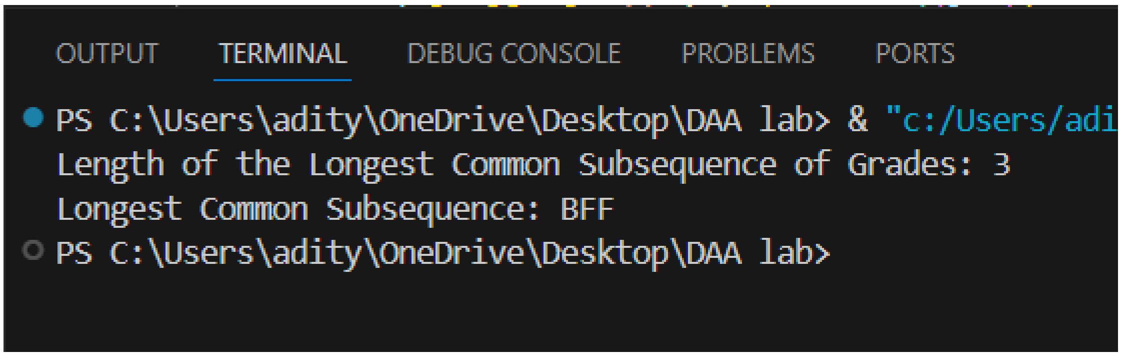
O(n.m) + O(m+n)
= O(n.m)

Time complexity = O(n·m)

Code:

```python
def longest_common_subsequence(s1, s2):
    # Initialize DP table with dimensions (len(s1)+1) x (len(s2)+1)
    dp = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]

    # Fill the DP table
    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
            if s1[i - 1] == s2[j - 1]:  # Matching characters
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:  # Take maximum of ignoring one character from either string
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # Trace back to find the LCS
    i, j = len(s1), len(s2)
    lcs = []
    while i > 0 and j > 0:
        if s1[i - 1] == s2[j - 1]:  # Characters match
            lcs.append(s1[i - 1])
            i -= 1
            j -= 1
        elif dp[i - 1][j] > dp[i][j - 1]:  # Move up in the DP table
            i -= 1
        else:  # Move left in the DP table
            j -= 1

    return dp[-1][-1], ''.join(reversed(lcs))  # LCS length and the
subsequence itself

# Example grades for 20 students
students_grades = [
    "AABBFF", "AABFFF", "AABBFF", "AABBFF", "AABFFF",
    "BBCCFF", "AABBFF", "AABFFF", "BBCCFF", "AABBFF",
    "AABBFF", "AABFFF", "AABBFF", "AABBFF", "AABFFF",
    "BBCCFF", "AABBFF", "AABFFF", "BBCCFF", "AABBFF"
]

# Compute LCS among all students
lcs_length, lcs_string = len(students_grades[0]), students_grades[0]
for i in range(1, len(students_grades)):
    lcs_length, lcs_string = longest_common_subsequence(lcs_string,
students_grades[i])

print(f"Length of the Longest Common Subsequence of Grades: {lcs_length}")
print(f"Longest Common Subsequence: {lcs_string}")
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c:/Users/adi
Length of the Longest Common Subsequence of Grades: 3
Longest Common Subsequence: BFF
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

Matrix Chain Multiplication (MCM) is an optimization problem where the goal is to determine the most efficient way to multiply a sequence of matrices. The order of multiplication significantly affects the computation cost, as matrix multiplication is not commutative or associative.

In the context of meteorological data, you are processing a two-dimensional array for parameters like temperature, dew point, wind speed, etc., for a week. Multiplying these matrices in the optimal order ensures timely predictions

Key Terms in Matrix Chain Multiplication

1. **Matrices**:
    - Each matrix represents meteorological data for a city or a day in terms of attributes (rows and columns). The dimensions of matrices need to be compatible for multiplication.

2. **Cost of Multiplication**:
    - The goal is to minimize the total multiplication cost for a sequence of matrices.

3. **Parenthesization**:
    - Different ways of grouping matrices for multiplication. For instance:
        - (A1·A2)·A3

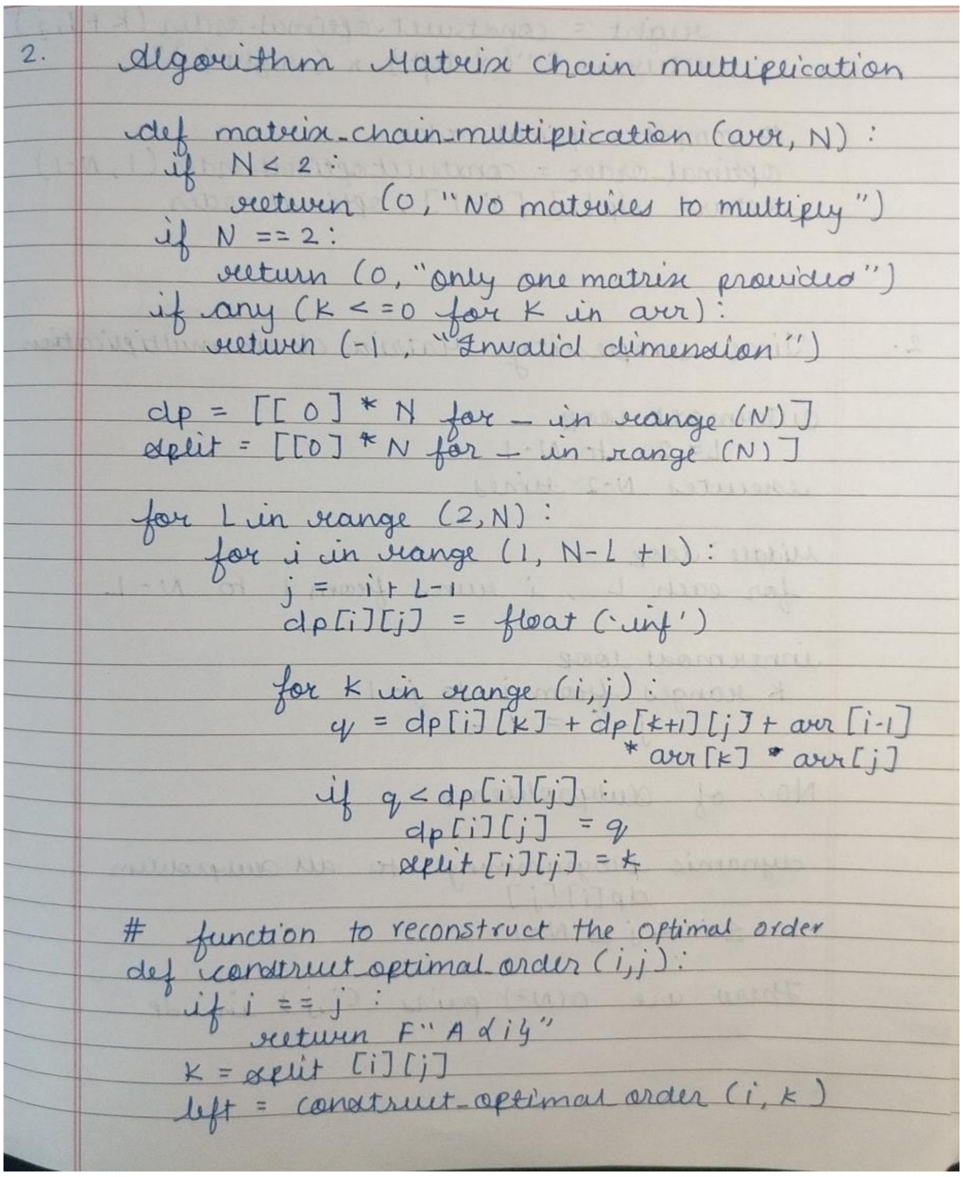
        ▪ A1·(A2·A3)

    o The multiplication order affects computation costs.

4. **Dynamic Programming Table (DP Table)**:
    o A 2D table dp[i][j] where:
        ▪ dp[i][j] represents the minimum cost to multiply matrices from I to j.

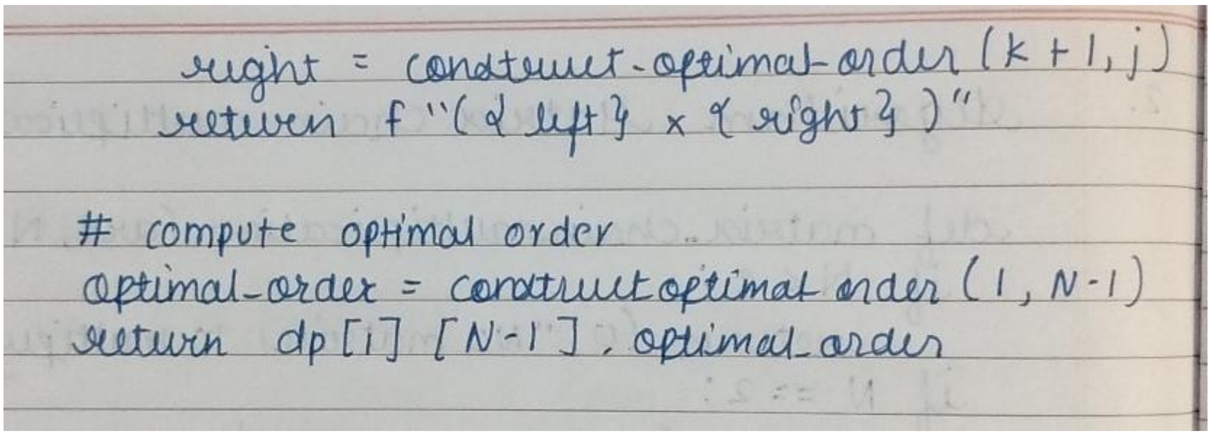
The table is filled iteratively to find the optimal solution

Algorithm Matrix chain multiplication:



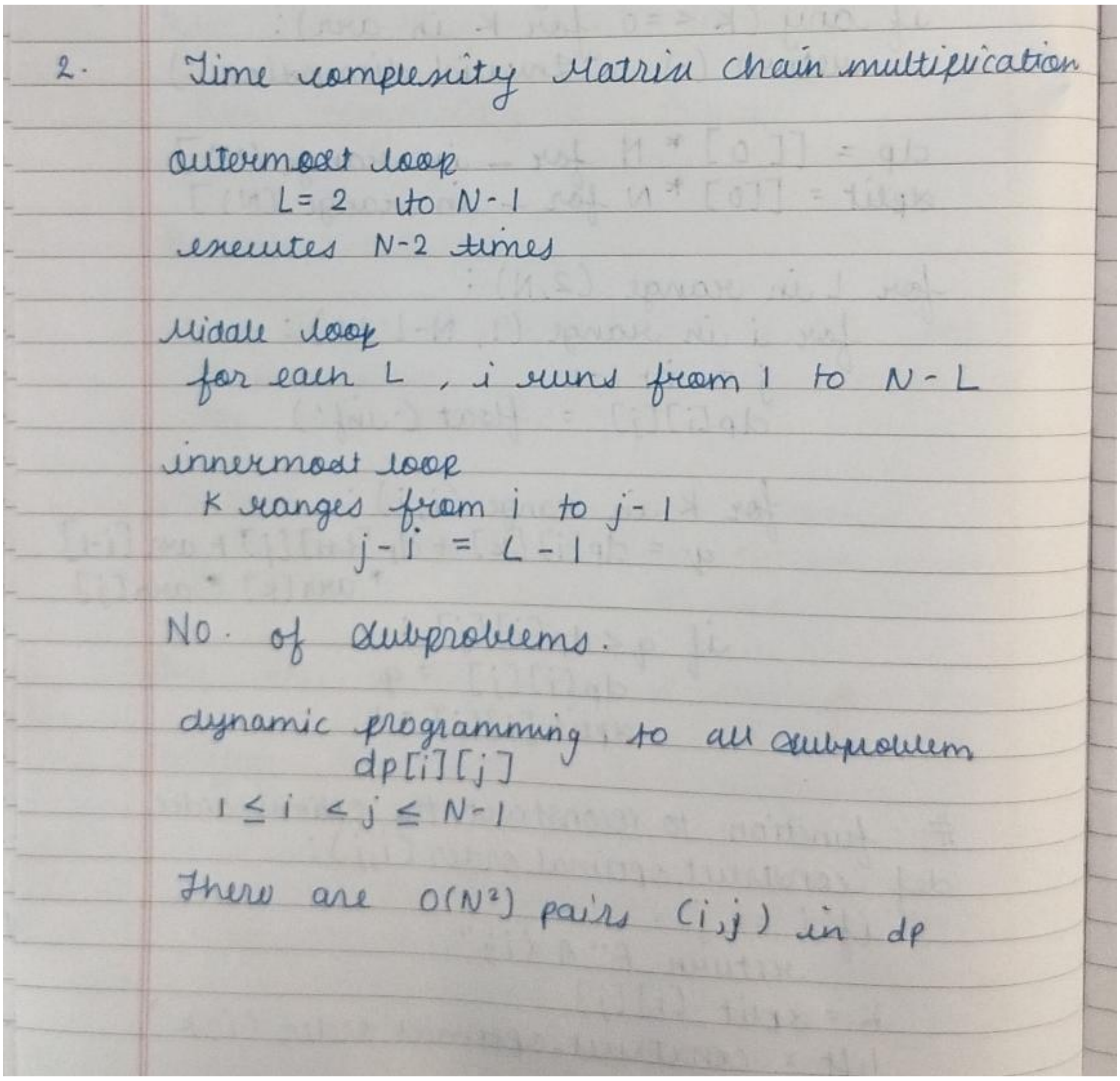
2. Algorithm Matrix chain multiplication

```
def matrix-chain-multiplication (arr, N) :
  if N < 2 :
     return (0, "NO matrices to multiply")
  if N == 2:
     return (0, "only one matrix provided")
  if any (k <=0 for k in arr):
     return (-1 , "Invalid dimension")

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

# function to reconstruct the optimal order
def construct_optimal_order (i,j) :
  if i == j :
     return F"A{i}"
  k = split [i] [j]
  left = construct_optimal_order (i, k)
```



```
        right = construct-optimal-order (k + 1, j)
        return f"({left} x {right})"

    # compute optimal order
    optimal-order = construct optimal order (1, N-1)
    return dp[1] [N-1], optimal-order
```

Time complexity Matrix chain multiplicaton:



2. Time complexity Matrix chain multiplication

outermost loop
L = 2 to N-1
executes N-2 times

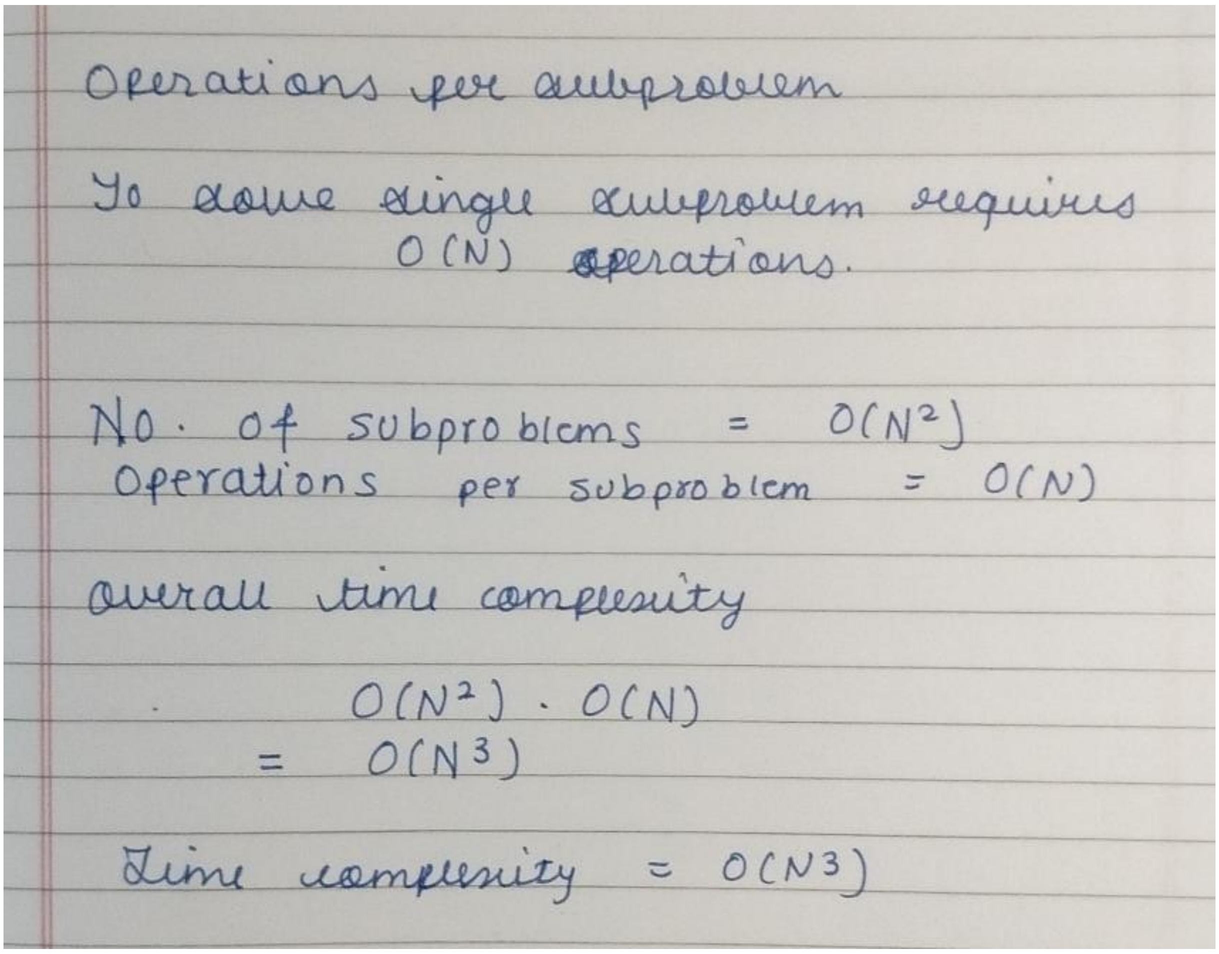
middle loop
for each L, i runs from 1 to N-L

innermost loop
K ranges from i to j-1
$j - i = L - 1$

No. of subproblems.

dynamic programming to all subproblem
dp[i][j]
$1 \le i < j \le N-1$

There are $O(N^2)$ pairs (i,j) in dp



Operations per subproblem

To solve single subproblem requires $O(N)$ operations.

No. of subproblems $= O(N^2)$
Operations per subproblem $= O(N)$

overall time complexity

$$O(N^2) \cdot O(N)$$
$$= O(N^3)$$

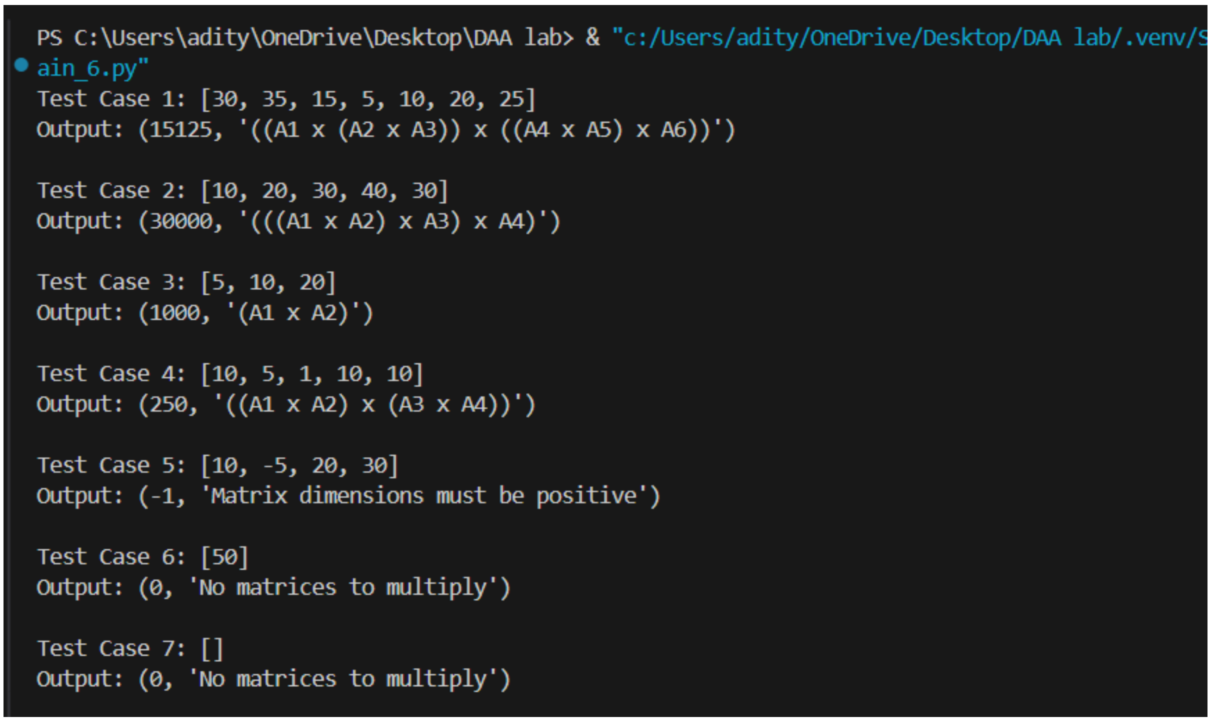
Time complexity $= O(N3)$

CODE:

```
def matrix_chain_multiplication(N, arr):
    """
    Computes the optimal order of multiplying matrices to minimize the number
of scalar multiplications.
    """
    if N < 2:
        return 0, "No matrices to multiply"  # Not enough dimensions for
multiplication
    if N == 2:
        return 0, "Only one matrix provided"  # A single matrix cannot be
multiplied
    if any(k <= 0 for k in arr):
        return -1, "Matrix dimensions must be positive"  # Invalid matrix
dimensions

    # DP and split tables
    dp = [[0] * N for _ in range(N)]
    split = [[0] * N for _ in range(N)]
```

```python
    # Compute the minimum cost for chain lengths L
    for L in range(2, N):  # L is the chain length
        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

    # Helper function to reconstruct the optimal order
    def construct_optimal_order(i, j):
        if i == j:
            return f"A{i}"
        k = split[i][j]
        left = construct_optimal_order(i, k)
        right = construct_optimal_order(k + 1, j)
        return f"({left} x {right})"

    # Compute the optimal order
    optimal_order = construct_optimal_order(1, N - 1)
    return dp[1][N - 1], optimal_order

# Test cases
def run_tests():
    """
    Run tests for matrix chain multiplication with different test cases.
    """
    test_cases = [
        [30, 35, 15, 5, 10, 20, 25],  # Classic case
        [10, 20, 30, 40, 30],          # Multiple matrices
        [5, 10, 20],                   # Small number of matrices
        [10, 5, 1, 10, 10],            # Different dimensions
        [10, -5, 20, 30],              # Negative dimensions
        [50],                          # Single dimension
        [],                            # No dimensions
    ]

    for idx, tc in enumerate(test_cases, 1):
        result = matrix_chain_multiplication(len(tc), tc)
        print(f"Test Case {idx}: {tc}")
        print(f"Output: {result}\n")

run_tests()
```

OUTPUT:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c:/Users/adity/OneDrive/Desktop/DAA lab/.venv/S
ain_6.py"
Test Case 1: [30, 35, 15, 5, 10, 20, 25]
Output: (15125, '((A1 x (A2 x A3)) x ((A4 x A5) x A6))')

Test Case 2: [10, 20, 30, 40, 30]
Output: (30000, '(((A1 x A2) x A3) x A4)')

Test Case 3: [5, 10, 20]
Output: (1000, '(A1 x A2)')

Test Case 4: [10, 5, 1, 10, 10]
Output: (250, '((A1 x A2) x (A3 x A4))')

Test Case 5: [10, -5, 20, 30]
Output: (-1, 'Matrix dimensions must be positive')

Test Case 6: [50]
Output: (0, 'No matrices to multiply')

Test Case 7: []
Output: (0, 'No matrices to multiply')
```

SOLID PRINCIPLES OF SOFTWARE DEVELOPMENT :

SOLID is an acronym that stands for:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

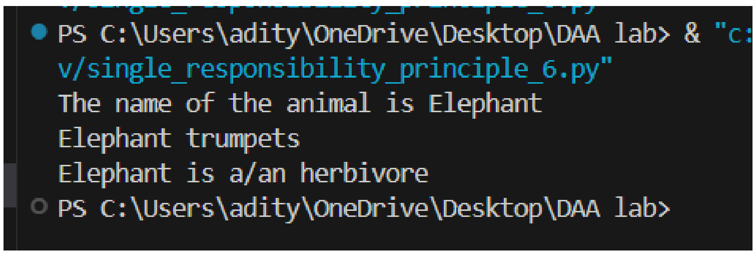
1. The Single Responsibility Principle (SRP)

The single responsibility principle states that a class, module, or function should have only one reason to change, meaning it should do one thing.

Code:

```python
# Animal class
class Animal:
    def __init__(self, name):
        self.name = name

    def nomenclature(self):
        print(f"The name of the animal is {self.name}")

# Sound class
class Sound:
    def __init__(self, name, sound_made):
        self.name = name
        self.sound_made = sound_made

    def sound(self):
        print(f"{self.name} {self.sound_made}s")

# Feeding class
class Feeding:
    def __init__(self, name, feeding_type):
        self.name = name
        self.feeding_type = feeding_type

    def feeding(self):
        print(f"{self.name} is a/an {self.feeding_type}")


# Create instances and call the methods
animal1 = Animal("Elephant")
animal1.nomenclature()  # The name of the animal is Elephant

animal_sound1 = Sound("Elephant", "trumpet")
animal_sound1.sound()  # Elephant trumpets

animal_feeding1 = Feeding("Elephant", "herbivore")
animal_feeding1.feeding()  # Elephant is a/an herbivore
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c:
v/single_responsibility_principle_6.py"
The name of the animal is Elephant
Elephant trumpets
Elephant is a/an herbivore
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

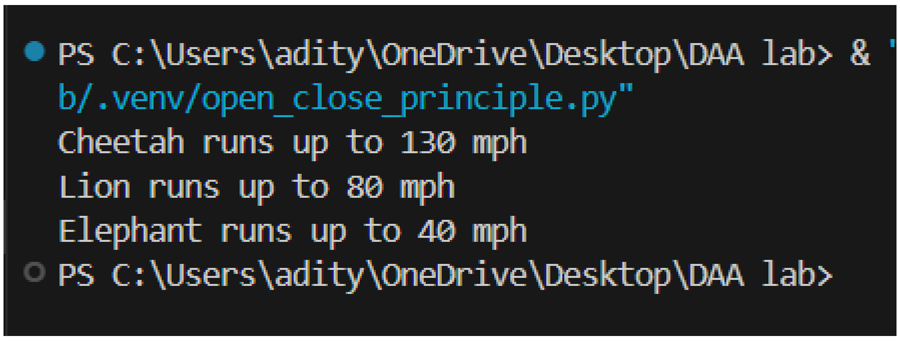
2. The Open-Closed Principle (OCP)

The open-closed principle states that classes, modules, and functions should be open for extension but closed for modification.

Code:

```
# Abstract SpeedRate class
class SpeedRate:
    def get_speed(self):
        raise NotImplementedError("This method should be overridden by 
subclasses")

# Concrete SpeedRate classes
class CheetahSpeedRate(SpeedRate):
    def get_speed(self):
        return 130

class LionSpeedRate(SpeedRate):
    def get_speed(self):
        return 80

class ElephantSpeedRate(SpeedRate):
    def get_speed(self):
        return 40

# Animal class
class Animal:
    def __init__(self, name, age, speed_rate):
        self.name = name
        self.age = age
        self.speed_rate = speed_rate

    def get_speed(self):
        return self.speed_rate.get_speed()


# Create instances of animals with different speed rates
cheetah = Animal("Cheetah", 4, CheetahSpeedRate())
print(f"{cheetah.name} runs up to {cheetah.get_speed()} mph")  # Cheetah runs 
up to 130 mph

lion = Animal("Lion", 5, LionSpeedRate())
print(f"{lion.name} runs up to {lion.get_speed()} mph")  # Lion runs up to 80 
mph

elephant = Animal("Elephant", 10, ElephantSpeedRate())
print(f"{elephant.name} runs up to {elephant.get_speed()} mph")  # Elephant 
runs up to 40 mph
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & '
b/.venv/open_close_principle.py"
Cheetah runs up to 130 mph
Lion runs up to 80 mph
Elephant runs up to 40 mph
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

3. The Liskov Substitution Principle (LSP)

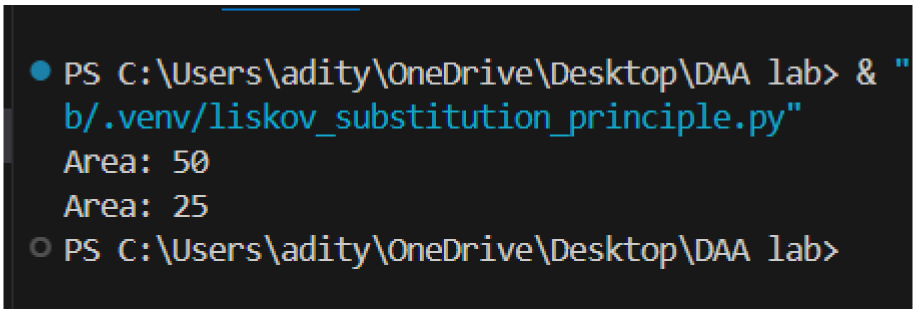
The principle states that child classes or subclasses must be substitutable for their parent classes or super classes. In other words, the child class must be able to replace the parent class.

Code:

```python
# Base class: Shape
class Shape:
    def area(self):
        pass

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

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

# Subclass: Square (no longer extends Rectangle, but still is a Shape)
class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length * self.side_length

# Test with LSP followed
def print_area(shape: Shape):
    print(f"Area: {shape.area()}")

# Create instances
rectangle = Rectangle(5, 10)
```

```python
square = Square(5)

print_area(rectangle)  # Works fine
print_area(square)  # Works fine
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "
b/.venv/liskov_substitution_principle.py"
Area: 50
Area: 25
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

4. The Interface Segregation Principle (ISP)

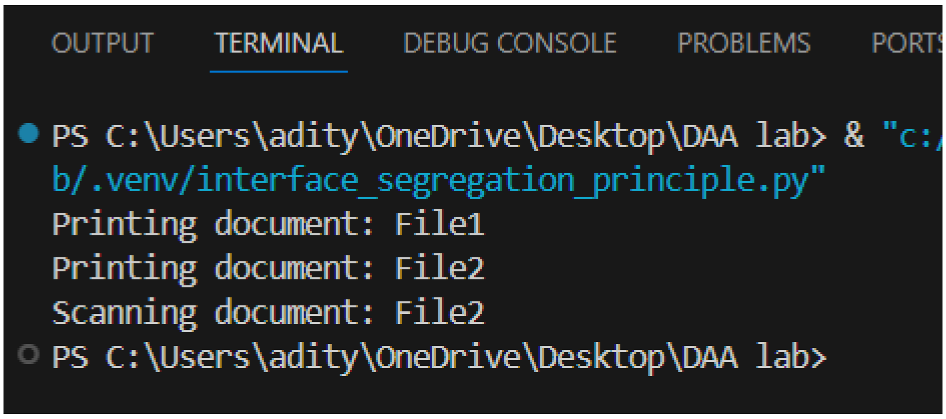
The interface segregation principle states that clients should not be forced to implement interfaces or methods they do not use.

Code:

```python
# Correct implementation of ISP: Smaller, more specific interfaces
class Printer:
    def print_document(self, document):
        pass

class Scanner:
    def scan_document(self, document):
        pass

# Class that only prints
class SimplePrinter(Printer):
    def print_document(self, document):
        print(f"Printing document: {document}")

# Class that can print and scan
class MultiFunctionPrinter(Printer, Scanner):
    def print_document(self, document):
        print(f"Printing document: {document}")

    def scan_document(self, document):
        print(f"Scanning document: {document}")

# Example usage
```

```
printer1 = SimplePrinter()
printer1.print_document("File1")  # This works fine
# printer1.scan_document("File1")  # No longer exists in SimplePrinter, so no
error

printer2 = MultiFunctionPrinter()
printer2.print_document("File2")  # This works fine
printer2.scan_document("File2")   # This works fine
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c:/
b/.venv/interface_segregation_principle.py"
Printing document: File1
Printing document: File2
Scanning document: File2
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

5. The Dependency Inversion Principle (DIP)

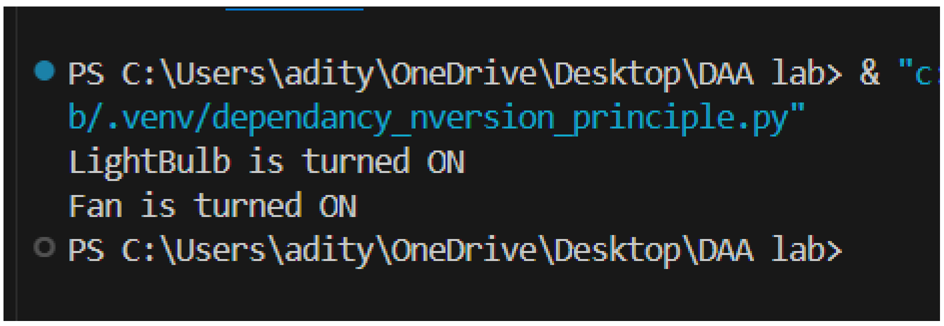
The dependency inversion principle is about decoupling software modules. That is, making them as separate from one another as possible, this principle suggests that instead of high-level modules directly depending on low-level modules, both should depend on abstract interfaces

Code:

```
# Abstraction: Switchable
class Switchable:
    def turn_on(self):
        pass

    def turn_off(self):
        pass


# Low-level module: LightBulb
class LightBulb(Switchable):
    def turn_on(self):
        print("LightBulb is turned ON")
```

```python
    def turn_off(self):
        print("LightBulb is turned OFF")


# Another low-level module: Fan
class Fan(Switchable):
    def turn_on(self):
        print("Fan is turned ON")

    def turn_off(self):
        print("Fan is turned OFF")


# High-level module: Switch
class Switch:
    def __init__(self, device: Switchable):
        self.device = device  # Dependency Injection

    def operate(self):
        self.device.turn_on()


# Test
lightbulb = LightBulb()
fan = Fan()

switch1 = Switch(lightbulb)  # Injecting LightBulb
switch1.operate()  # LightBulb is turned ON

switch2 = Switch(fan)  # Injecting Fan
switch2.operate()  # Fan is turned ON
```

Output:



```
PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c
b/.venv/dependancy_nversion_principle.py"
LightBulb is turned ON
Fan is turned ON
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

Conclusion : Hence in this practical we have learnt

1. We learnt and implemented Longest common subsequence Algorithm using Dynamic programming

2. We learnt and implement the matrix chain multiplication algorithm using dynamic programming

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