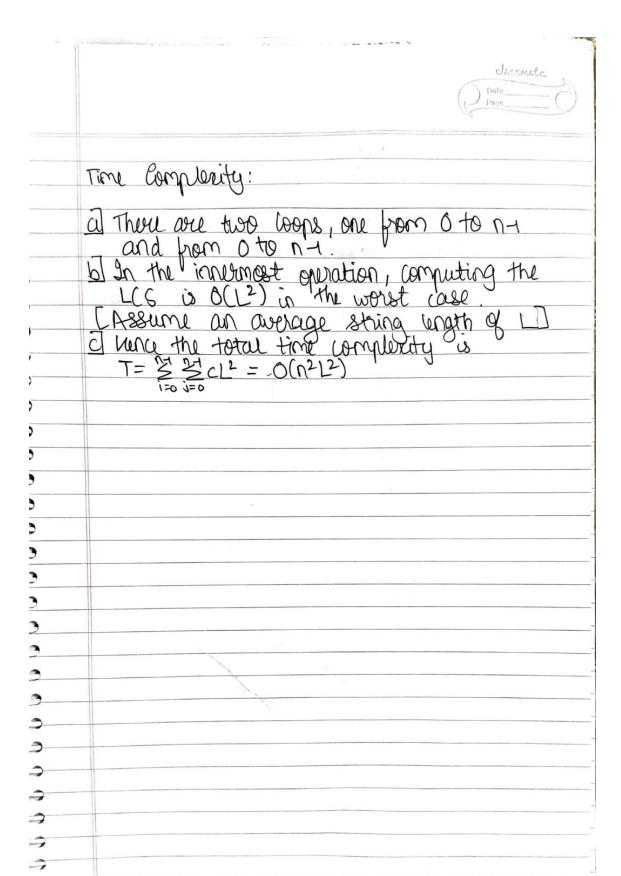
Experiment 06

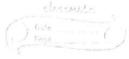
	Classmate Date Page
	Algorithm Find LCS (string SI, string S2)
)	1/ computes the languat common subsquences between 2 strings
)	11 Input: Two strings SI & SZ 11 Dutput: LCS of the strings
5	let m = S1. length and n = s2. length
))	declare a table of my nows and not
5	for i=1 to m: for j=1 to n: y si[i-] == s2[j-]:
3	table[i][j] = table[i][j-i] +1
3	table[i][j] = larger of table[i][j] and table[i][j] based
•	on their length.
<u> </u>	gretter table[m][n]
<u> </u>	Time Complexity:
	a let 'c' be the constant time taken for the innormast appration in the loops. The first loop suins from 1 to m and 2nd loop suins from 1 to n.
- -	be the first loop. The strong I to m and 21d
-	



	~ ^
-	J True T = Z Z C
	1=1 J=1
-	$T = M\Omega C$
	T = mnC $T = 0(mn)$
	Hence, time complexity is O(mn) where m&n are the lengths of the string.
	are the lengths of the string.
	They book sower than to be with a second
*	Algorithm Find LCS Multiple (string s).
	11 Compute the LCS among n strings 11 Imput: Array of n strings 11 Output: LCS of n strings
	11 Input: Array of n strings
	11 Dulput: LCS of n strings.
	The state of the s
	18x 1 - 0 to 07:
	Seq = " "
- 1	for j=0 to n-1:
	if it is
	LCS = FINALLS (LCS, SGJ)
	y US==
	Greak
	sea = moncl sea, LCS,
	seq = moncl seq, LCS, Key = length).
	notion seg



)



3	Costs Const
٥ ٥	Testcases:
	a] ["AABBCCDD", "BBCCDAA"]
) 3——	€ ["BCBCCDFF", "FFBCAADDI]
))	J ["ABABOOCD", "FFDOFFCO"] → DDCO
•	["ARAAAAA", "ABBBBBBB"]
	o] ["PFFFFFFF", "FFFFFFF"] → FFFFFFF
))	I ["", "AABBCCODFF"] → no char in string
)))	al["BOFFCCAAOD", "RADDFF34"] string should not contain numbers
)))	h]["ARAAAAAAA", "BBBBBB"] no common subsequence found
))	i] ["AABBODFFAA", "AABBFF440p"] -> 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: FFFFFFF

Negative Test Cases:

Test Case 6: No characters in the string

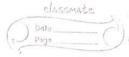
Test Case 7: Grade sequences should not contain numbers

Test Case 8: No common subsequence found

Test Case 9: Grade sequences should not contain numbers

23

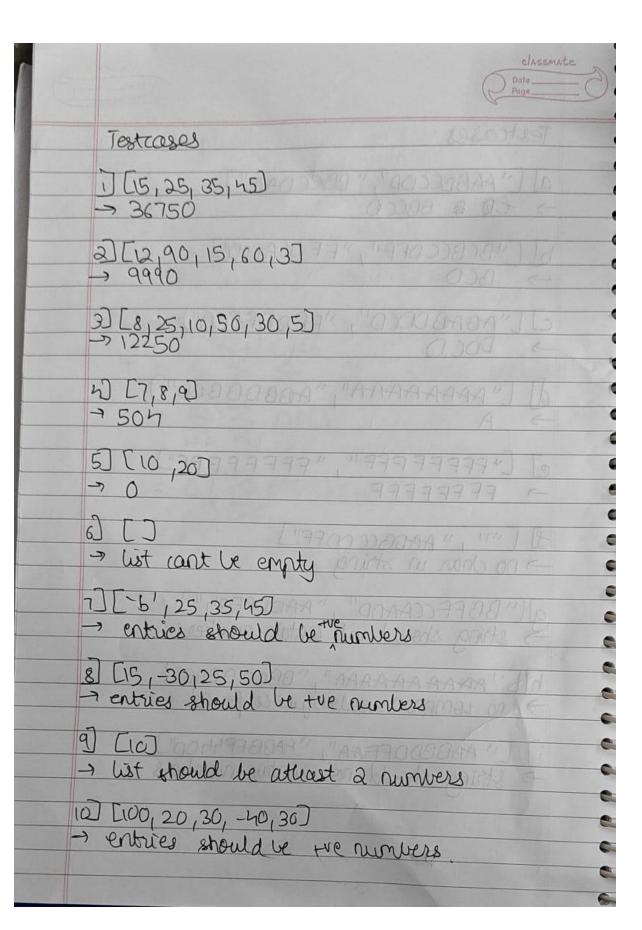
)	ALL SELLING
`	classmate
	Page
)	Mass de la company de la compa
	Algorithm Matrix Chain Multiplication (N, are): 11 Input: An array containing the matrix demensions.
	11 Input: An array containing the matrix
	Unit to a superior and another incition
2)	11 Output: smallest number of multiplication and optimal order.
-	as ye of the face of the s.
	delico dell'Ille Das a 20 apparente
-	define de [1
)	A STATE OF THE STA
)	define S[1N] as the split points.
)	Applies NUMBER (1970)
.	101 L=2 to N-1:
~	for i=1 to N-L:
ə	$j = i + L - 1$ $dp \leq j \leq j \leq 3$
-	for K=1 to j=1:
•	a = dotiJCIZ + do(k+iJCiJ
>	q=dpti][i]+dp[k+i][j] +an[i-i]*an[k]*an[j]
>	y q < dp[i][i]:
3	COLULY - 9,
3	SCiJCiJ = K
3	optional-order = Get optimalerder (1, N-1)
2	
3	geturn dp[][N-1], optimal-order
-	
,	getoptimalorder(i,j):
	o 'y == y return "M" +i
· 	K = Split Listy
-	right = petantionalism as (v. 17
7	getoptimalorder(i,j): \[\i = j \teturn \mathbb{M}' + i \\ \k = &p\lit CiJCjJ \\ \light = Getoptimulorder (i,k) \\ \text{right} = \text{getoptimalorder}(\kmi,j) \\ \text{right} \left(\left) \text{right} \left(\text{right} \reft) \\ \text{right} \left(\left) \text{right} \reft(\text{right} \reft) \\ \text{right} \left(\text{right} \reft) \\ \text{right} \text{right} \reft(\text{right} \reft) \\ \text{right} \text{right} \right(\text{right} \right) \\ \text{right} \text{right} \right) \\ \text{right} \right(\text{right} \right) \\ \text{right} \right
-	Taran Capana



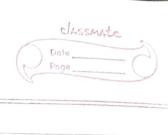
Time Complexity:
a) There are 3 nested loops in the algorithm.
algorithm. • outer loop: L=2 to N-1
• Outer Coop: L=2 to N-L • middle loop: i=1 to N-L • inner loop: k=i to i+L-2
b) surring up the loop operations.
$T = \underbrace{\sum_{i=1}^{N-L} \sum_{k=1}^{N+L-2} O(1)}_{L=2} \underbrace{\underbrace{\sum_{i=1}^{N-L-1} O(1)}_{L=2}}_{O(L)} \underbrace{\underbrace{\underbrace{\sum_{i=1}^{N-L-1} \sum_{i=1}^{N-L-1} O(1)}_{L=2}}_{I=2}}_{I=1}$
= \$ 0(L)
$= 0 \left(\frac{N^2}{2} (N-L)(L) \right)$
$= O\left(N \leq L - \leq L^2\right)$

$$\approx 0 \left(\frac{NN^2 - N^3}{6} \right)$$

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



```
O ---- 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 largest common subsequence algorithm using dynamic programming and found the largest common subsequence for 20 sequences of grades.

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

SOLID Branciples using sample classes and objects.

0.300

3

3

n

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):
        \overline{\text{self.width}} = \text{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.

```
# 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.