Name: Bhakti Anil Baraf
ID: 241071907
B.Tech DSY Computer Engineering

Practical No. 6

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.

Steps for lcs(X, Y):

1. **Validation**:
    - Check if both X and Y are strings. If not, raise an exception.
2. **Create a 2D DP table**:
    - Let dp[i][j] represent the length of LCS of X[0...i-1] and Y[0...j-1].
    - The dimensions of the table will be (m + 1) x (n + 1), where m and n are the lengths of strings X and Y respectively.
3. **Fill the DP Table**:
    - Loop through each character pair from X and Y, filling in the table based on character matches or the maximum of previous results.
4. **Reconstruct the LCS**:
    - Start from the bottom-right of the DP table and trace back to the top-left, appending characters when they match.
5. **Return the result**:
    - The LCS is reconstructed and reversed, as the backtracking builds the sequence in reverse order.

Algorithm:

Lab 6.

DATE:

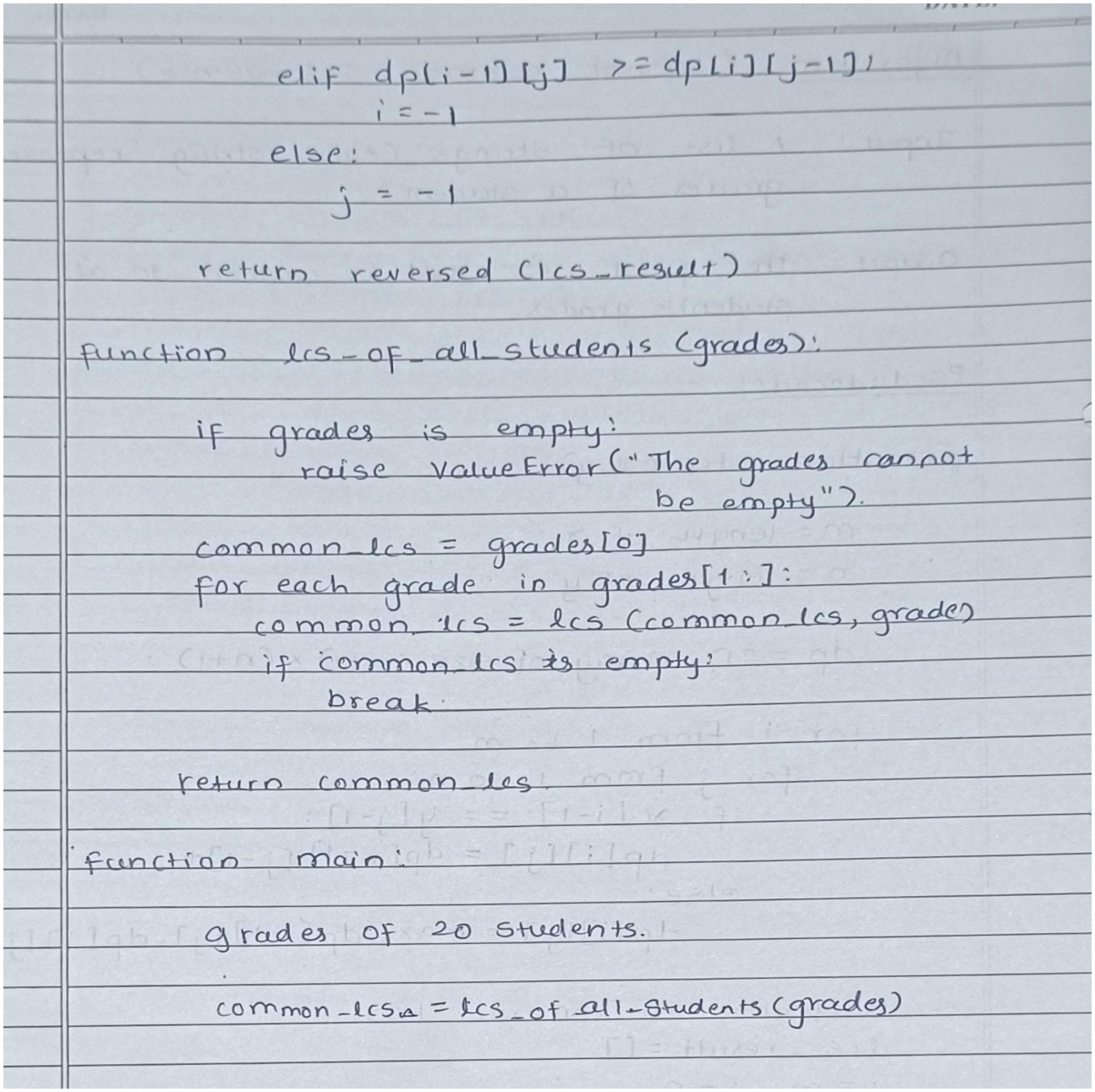
Algorithm : (Longest common subsequence)

Input : A list of strings (each string represents grades of a student)

Output : The final LCS that is common to all student's grades.

Pseudocode:

```
function lcs(x, y):

    m = length of x
    n = length of y

    dp = 2D array of size (m+1) x (n+1).

    for i from 1 to m:
        for j from 1 to n:
            if x[i-1] == y[j-1]:
                dp[i][j] = dp[i-1][j-1]+1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    Backtract to construct LCS.
    lcs_result = []
    i = m, j = n
    while i >0 and j >0:
        if x[i-1] == y[j-1]:
            append x[i-1] to lcs_result.
            i = -1,
            j = -1
```



Code:

```
def lcs(X, Y):
    try:
        # Validate inputs
        if not isinstance(X, str) or not isinstance(Y, str):
            raise ValueError("Both input sequences must be strings.")
```

```
    m, n = len(X), len(Y)

    # Create a 2D table to store lengths of longest common subsequence.
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill the table in a bottom-up manner
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1  # If characters match, add 1 to the result from the previous diagonal
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])  # Otherwise, take the maximum of the left or top

    # Reconstruct the LCS from the dp table
    lcs_result = []
    i, j = m, n
    while i > 0 and j > 0:
        if X[i - 1] == Y[j - 1]:
            lcs_result.append(X[i - 1])  # Add the character to the result
            i -= 1
            j -= 1
        elif dp[i - 1][j] >= dp[i][j - 1]:
            i -= 1
        else:
            j -= 1
```

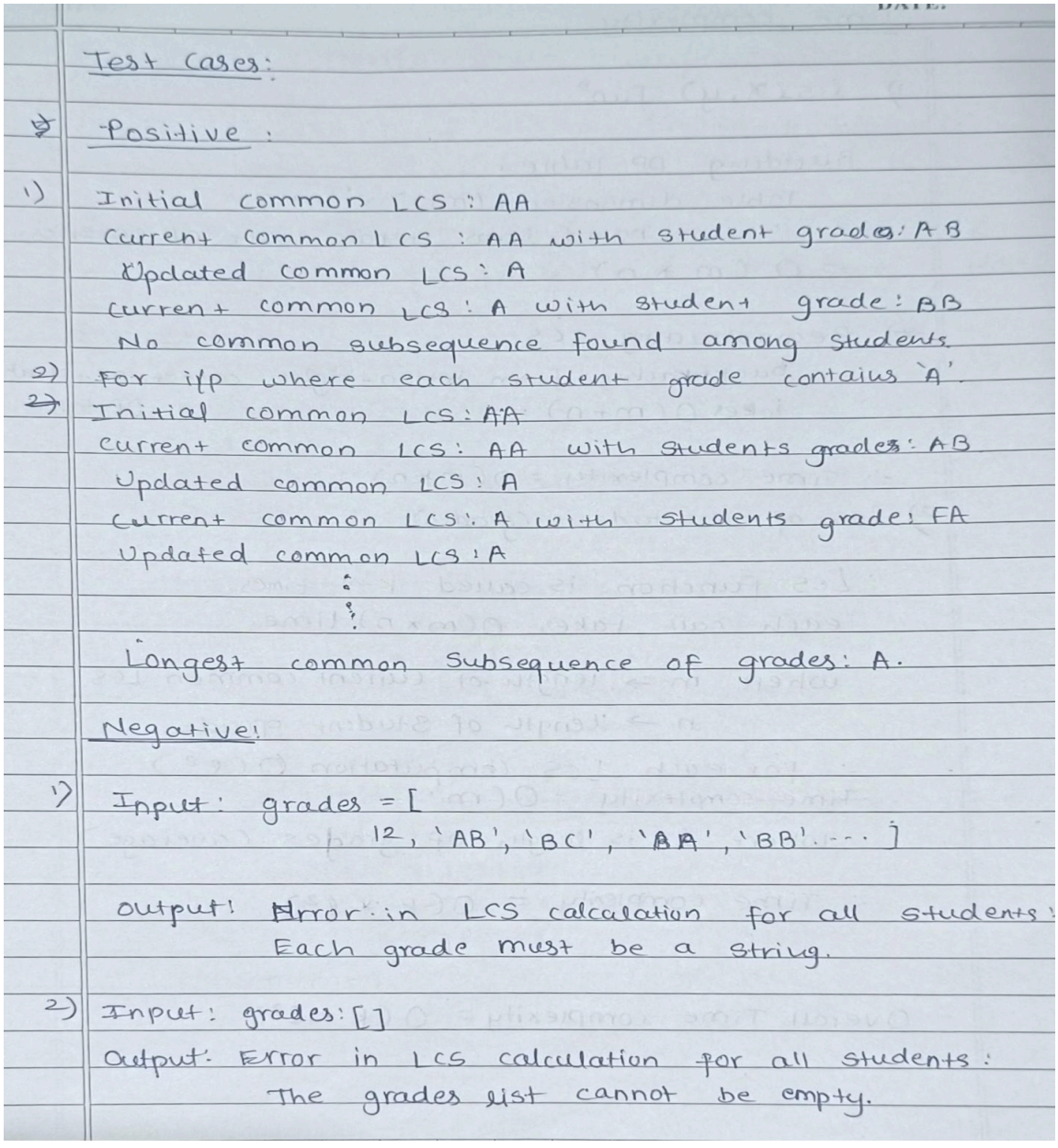
```
        # Return the LCS result as a string
        return ''.join(reversed(lcs_result))  # Reverse because we built the LCS backwards

    except Exception as e:
        print(f"Error in LCS calculation: {e}")
        return ""


def lcs_of_all_students(grades):
    try:
        # Validate input grades list
        if not grades:
            raise ValueError("The grades list cannot be empty.")

        if not all(isinstance(grade, str) for grade in grades):
            raise ValueError("Each grade must be a string.")

        # Start with the first student's grades
        common_lcs = grades[0]
        print(f"Initial common LCS: {common_lcs}")

        # Compute LCS with each subsequent student's grades
        for grade in grades[1:]:
            print(f"Current common LCS: {common_lcs} with student grade: {grade}")
            common_lcs = lcs(common_lcs, grade)
            if not common_lcs:  # If at any point, the LCS becomes empty, break early
                break
```

```
            print(f"Updated common LCS: {common_lcs}")

        return common_lcs

    except Exception as e:
        print(f"Error in LCS calculation for all students: {e}")
        return ""


def main():
        # Example grades for 20 students
        grades = [
            "AA", "AB", "BB", "BC", "CD", "DE", "EF", "FA", "AA", "BB",
            "AA", "BA", "CC", "AB", "EF", "BA", "BC", "CD", "AA", "AA"
        ]

        # Compute LCS for all students
        common_lcs = lcs_of_all_students(grades)

        if common_lcs:
            print(f"Longest Common Subsequence of grades: {common_lcs}")
        else:
            print("No common subsequence found among students.")

if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bh
Longest Common Subsequence of Grades: ['BB', 'CC', 'FF']
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bh
Initial common LCS: AA
Current common LCS: AA with student grade: AB
Updated common LCS: A
Current common LCS: A with student grade: BB
No common subsequence found among students.
```

Test Cases:



Test Cases:

Positive:

1) Initial common LCS: AA
Current common LCS: AA with student grades: AB
Updated common LCS: A
Current common LCS: A with student grade: BB
No common subsequence found among students.

2) For i/p where each student grade contains 'A'.

Initial common LCS: AA
Current common LCS: AA with students grades: AB
Updated common LCS: A
Current common LCS: A with students grade: FA
Updated common LCS: A
⋮

Longest common subsequence of grades: A.

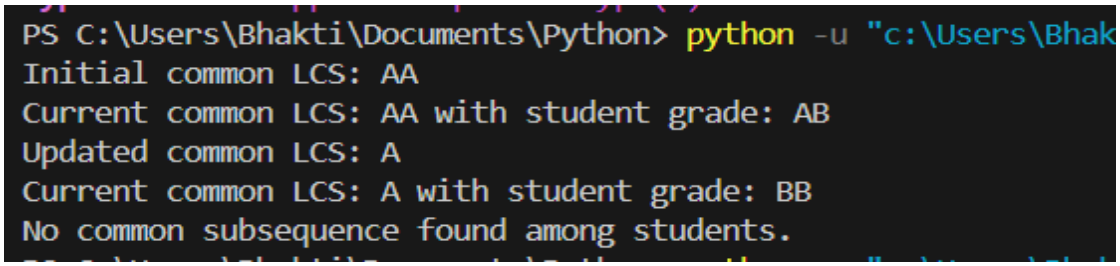
Negative:

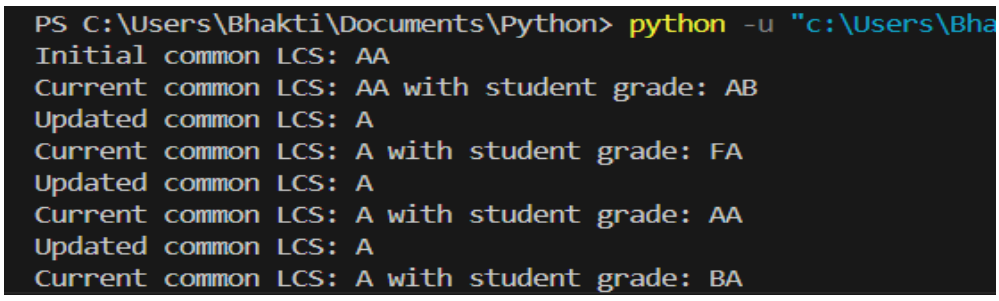
1) Input: grades = [
12, 'AB', 'BC', 'AA', 'BB'. . . .]

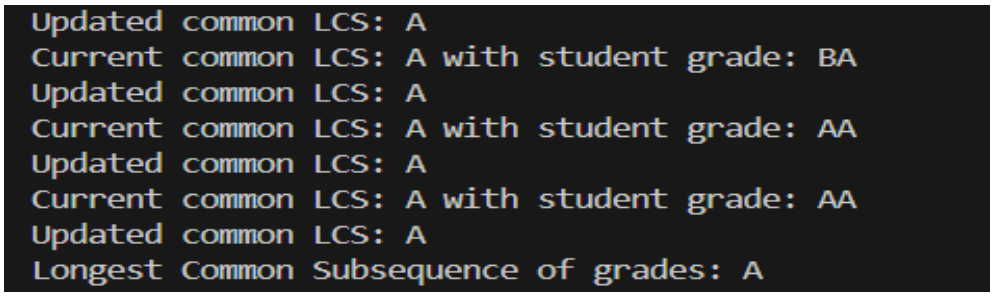
Output: Error in LCS calculation for all students:
Each grade must be a string.

2) Input: grades: []
Output: Error in LCS calculation for all students:
The grades list cannot be empty.

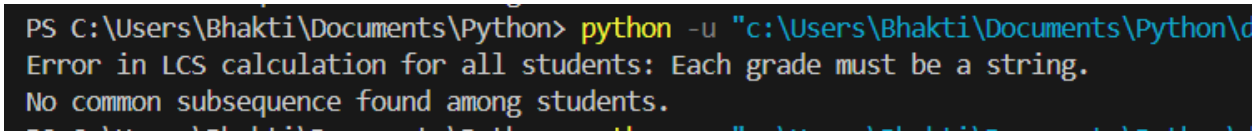
Positive Test Case:

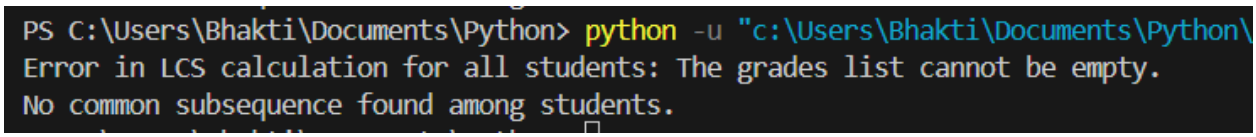




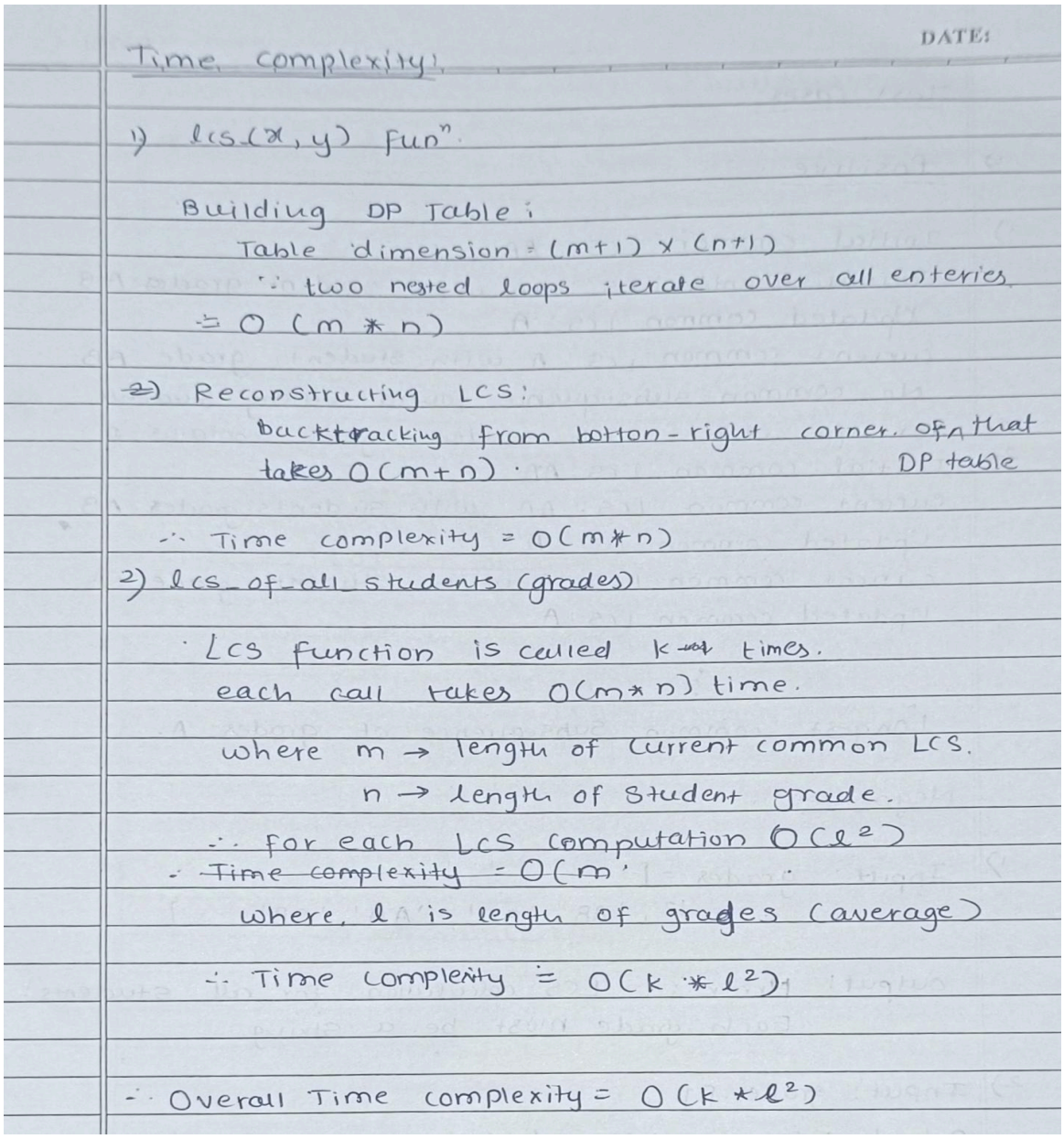


Negative Test Case:





Time Complexity:



Time complexity:

DATE:

1) lcs(x, y) fun^{n}.

Building DP Table:

Table dimension = (m+1) x (n+1)

∵ two nested loops iterate over all enteries.

≒ O (m * n)

2) Reconstructing LCS:

backtracking from botton-right corner of that DP table takes O(m+n).

∴ Time complexity = O(m*n)

2) lcs_of_all_students (grades)

· Lcs function is called k times.

each call takes O(m*n) time.

where m → length of current common LCS.

n → length of Student grade.

∴ for each LCS computation O(l^2)

~~Time complexity = O(m~~

where, l is length of grades (average)

∴ Time complexity ≒ O(k * l^2)

∴ Overall Time complexity = O(k * l^2)

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.

Algorithm:

DATE:

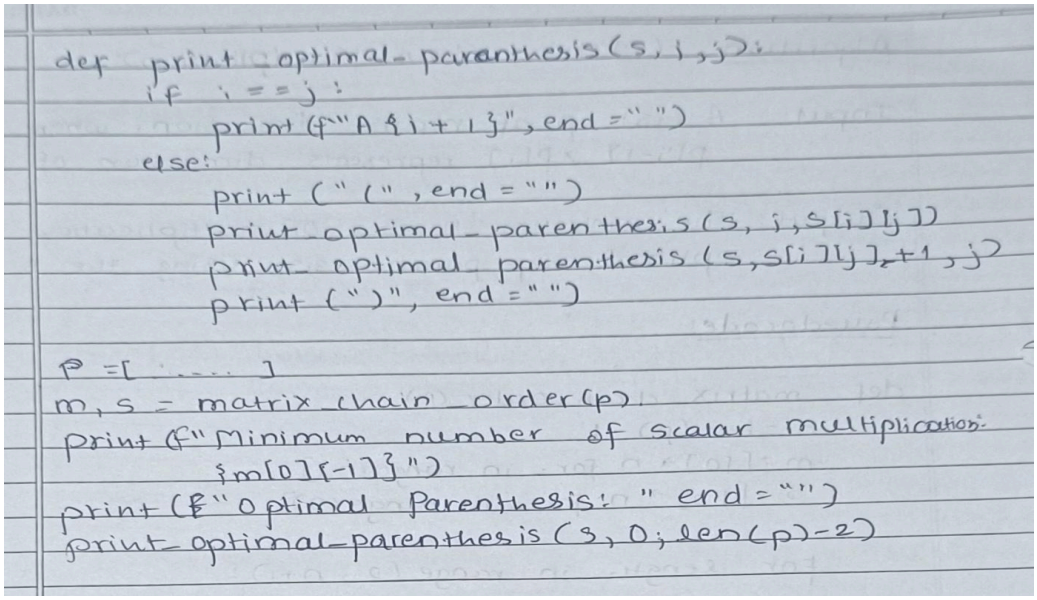
Algorithm : (Matrix Chain Multiplication):

Input: A list p of matrix dimension, where.
p[i-1] x p[i] represents dimension of matrix.

Output: The minimum number of multiplications & optimal ref paranthesis using the s table.

Psuedocode:

```
def matrix_chain_order (p):
    n = len (p) - 1
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]

    for length in range (2, n+1):
        for i in range (n - length + 1):
            j = i + length - 1
            m[i][j] = float ('inf')

    for k in range (i, j):
        q = m[i][k] + m[k+1][j] + p[i] * p[k+1] *
            p[j+1]
        if q < m[i][j]:
            m[i][j] = q
            s[i][j] = k.
    return m, s.
```



Code:

```
def matrix_chain_order(p):
    # Error handling for invalid input
    if not p or len(p) < 2:
        raise ValueError("Input list p must contain at least two elements representing matrix dimensions.")

    n = len(p) - 1  # Number of matrices

    # Initialize m and s tables
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]

    # l is the chain length
    for length in range(2, n + 1):  # length of chain from 2 to n
        for i in range(n - length + 1):  # i is the starting index of the chain
            j = i + length - 1  # j is the ending index of the chain

            m[i][j] = float('inf')  # Initialize to infinity
```

```
            # Try every possible split
            for k in range(i, j):
                # Calculate the cost of splitting at k
                q = m[i][k] + m[k + 1][j] + p[i] * p[k + 1] * p[j + 1]

                # If this is the minimal cost, update m[i][j]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k

    # Return the minimum number of multiplications and the split table
    return m, s

def print_optimal_parenthesization(s, i, j):
    if i == j:
        print(f"A{i+1}", end="")
    else:
        print("(", end="")
        print_optimal_parenthesization(s, i, s[i][j])
        print_optimal_parenthesization(s, s[i][j] + 1, j)
        print(")", end="")

p = [7, 5, 6, 4]  # Example matrix dimensions (can be modified as needed)

# Perform matrix chain multiplication
m, s = matrix_chain_order(p)

# Print the result
```

```
print(f"Minimum number of scalar multiplications: {m[0][-1]}")
print("Optimal parenthesization: ", end="")
print_optimal_parenthesization(s, 0, len(p) - 2)
```

Output:

```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documer
Minimum number of scalar multiplications: 260
Optimal parenthesization: (A1(A2A3))
```

Test Cases:

Test Cases: DATE:

Positive:

1) Input: p = [7, 5, 6, 4]

Output: Minimum number of scalar multiplication: 260
Optimal parenthesis: (A1(A2A3))

2) Input: p[7, 5, 10, 2, 3]

Output: Minimum number of scalar multiplication: 215
Optimal parenthesis: ((A1(A2A3))A4).

Negative:

1) ~~p = []~~ Input: p = []
Output: R
Error: Input list p must contain at least two elements representing matrix dimensions.

2) Input: p = [4]

Output:
Error: Input list p must contain at least two elements representing matrix dimensions.

Positive Test Case:

```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documen
Minimum number of scalar multiplications: 260
Optimal parenthesization: (A1(A2A3))
```

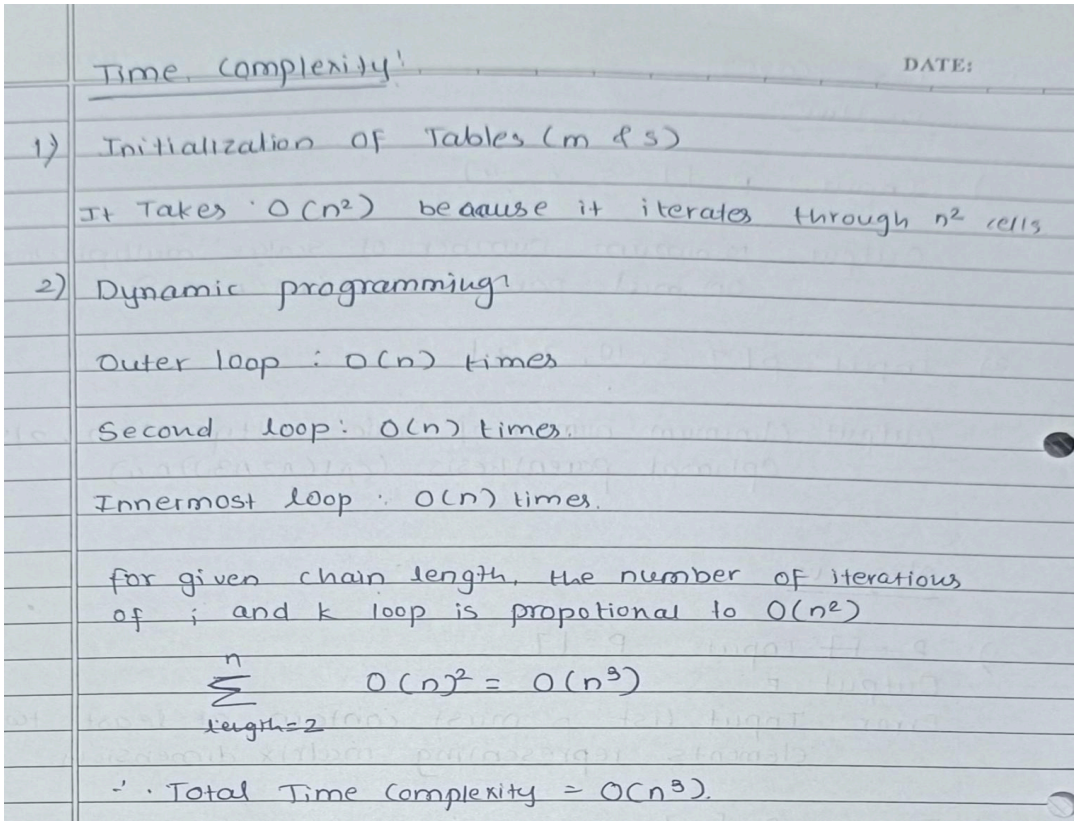
```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documen
Minimum number of scalar multiplications: 212
Optimal parenthesization: ((A1(A2A3))A4)
```

Negative Test Case:

```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documents\Python\daa_lab6_task
Error: Input list p must contain at least two elements representing matrix dimensions.
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documents\Python\tempCodeRunne
```

```
PS C:\Users\Bhakti\Documents\Python> python -u "c:\Users\Bhakti\Documents\Python\daa_lab6_task2.py"
Error: Input list p must contain at least two elements representing matrix dimensions.
PS C:\Users\Bhakti\Documents\Python>
```

TimeComplexity:



Time complexity:

DATE:

1) Initialization of Tables (m & s)

It Takes $O(n^2)$ because it iterates through n^2 cells

2) Dynamic programming

Outer loop : $O(n)$ times

Second loop: $O(n)$ times.

Innermost loop : $O(n)$ times.

for given chain length, the number of iterations of i and k loop is propotional to $O(n^2)$

$$\sum_{length=2}^{n} O(n)^2 = O(n^3)$$

∴ Total Time Complexity = $O(n^3)$.

Conclusion:

In this experiment we implemented algorithm for longest common sequence and matrix chain multiplication