**ASSIGNMENT 1**

You are given an array of integers, and you are required to sort this array using one of the following sorting algorithms: Bubble Sort, Selection Sort, or Insertion Sort. Your task is to implement the chosen sorting algorithm and analyze its time complexity.

1. Implement one of the sorting algorithms mentioned above (Bubble Sort, Selection Sort, or Insertion Sort) in Python.

2. Apply your sorting algorithm to the given array of integers.

3. Provide the sorted array as the output.

4. Analyze the time complexity of the sorting algorithm you implemented. Explain whether it is a stable sort and how it performs on different types of input data (e.g., already sorted, reverse sorted, random data).

5. Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm (e.g., Quick Sort, Merge Sort, or Python's built-in sorted function). Explain the differences and scenarios where one algorithm might be preferred over the other.

Input:

• An unsorted list of integers (e.g., [5, 2, 9, 1, 5, 6]).

Output:

• The sorted list of integers.

Instructions:

1. Choose one of the sorting algorithms (Bubble Sort, Selection Sort, or Insertion Sort) and implement it in Python.

2. Apply your chosen sorting algorithm to the provided input array.

3. Provide the sorted array as the output.

4. Analyze the time complexity of the sorting algorithm and discuss its stability and performance on different input data.

5. Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm, and explain when you would prefer one over the other.

Step 1: Implementing the Sorting Algorithm (I'll choose Bubble Sort)

python

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

Step 2 and 3: Applying the Sorting Algorithm to the Given Input

Input: [5, 2, 9, 1, 5, 6]

Sorted Output using Bubble Sort: [1, 2, 5, 5, 6, 9]

Step 4: Analyzing the Time Complexity, Stability, and Performance

- Time Complexity of Bubble Sort:

- Best-case time complexity: O(n)

- Average-case time complexity: O(n^2)

- Worst-case time complexity: O(n^2)

- Bubble Sort is not stable.

- Performance on different input data:

- Already Sorted: Bubble Sort has a best-case time complexity of O(n), making it efficient for already sorted data.

- Reverse Sorted: It performs with the worst-case time complexity of O(n^2), which is less efficient for reverse sorted data.

- Random Data: It has an average-case time complexity of O(n^2), which is not the most efficient for large datasets.

Step 5: Comparing with Another Sorting Algorithm (Let's choose Quick Sort)

Quick Sort:

- Average-case time complexity: O(n log n)

- Worst-case time complexity: O(n^2) (rare, but can happen)

- Quick Sort is not stable.

Comparison and Preferred Usage:

- Bubble Sort vs. Quick Sort:

- Quick Sort has better average-case and worst-case time complexities compared to Bubble Sort. It's generally more efficient for larger datasets.

- Bubble Sort might be preferred when simplicity of implementation is more important than performance, or when dealing with nearly sorted data.

In most cases, Quick Sort or other efficient sorting algorithms like Merge Sort would be preferred over Bubble Sort due to their better average-case performance. However, Bubble Sort can be suitable for small datasets or when simplicity is prioritized over efficiency.

**ASSIGNMENT 2**

You are given a sequence of matrices with dimensions that are suitable for matrix multiplication. Your task is to find the optimal way to parenthesize the matrices to minimize the total number of scalar multiplications required to compute their product.

1. Implement a dynamic programming algorithm in Python to solve the matrix chain multiplication problem.

2. Apply your algorithm to the given sequence of matrices and find the optimal parenthesization.

3. Calculate and provide the minimum number of scalar multiplications required for the optimal parenthesization.

4. Explain the dynamic programming approach you used, including the initialization, recurrence relation, and how you reconstructed the optimal parenthesization.

5. Analyze the time and space complexity of your algorithm, and discuss its efficiency in solving large instances of the problem.

Input:

• A list of matrices, each represented by its dimensions. For example, a list of matrices [A, B, C] could be represented as [(2, 3), (3, 4), (4, 2)] where the dimensions of matrix A are 2x3, the dimensions of matrix B are 3x4, and the dimensions of matrix C are 4x2.

Output:

• The optimal parenthesization of matrices as a sequence of matrix multiplications.

• The minimum number of scalar multiplications required for the optimal parenthesization.

Instructions:

1. Implement a dynamic programming algorithm to solve the matrix chain multiplication problem in Python.

2. Apply your algorithm to the provided list of matrices to find the optimal parenthesization.

3. Calculate and provide the minimum number of scalar multiplications required for the optimal parenthesization.

4. Explain the dynamic programming approach, including the initialization, recurrence relation, and reconstruction of the optimal parenthesization.

5. Analyze the time and space complexity of your algorithm and discuss its efficiency for large instances of the problem.

Step 1: Implementing the Dynamic Programming Algorithm

def matrix\_chain\_multiplication(dims):

n = len(dims) - 1

dp = [[float('inf')] \* n for \_ in range(n)]

for i in range(n):

dp[i][i] = 0

for chain\_length in range(2, n+1):

for i in range(n - chain\_length + 1):

j = i + chain\_length - 1

for k in range(i, j):

cost = dims[i][0] \* dims[k][1] \* dims[j][1]

dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + cost)

return dp[0][-1]

def optimal\_parenthesization(dims):

n = len(dims) - 1

dp = [[0] \* n for \_ in range(n)]

for i in range(n):

dp[i][i] = i

for chain\_length in range(2, n+1):

for i in range(n - chain\_length + 1):

j = i + chain\_length - 1

dp[i][j] = min(range(i, j), key=lambda k: dp[i][k] + dp[k+1][j] + dims[i][0] \* dims[k][1] \* dims[j][1])

return construct\_parenthesization(dp, 0, n-1)

def construct\_parenthesization(dp, i, j):

if i == j:

return f'M{str(i+1)}'

k = dp[i][j]

return f'({construct\_parenthesization(dp, i, k)} × {construct\_parenthesization(dp, k+1, j)})

Step 2 and 3: Applying the Algorithm to the Given Matrices

Input: [(2, 3), (3, 4), (4, 2)]

Optimal Parenthesization: ((M1 × M2) × M3)

Minimum Scalar Multiplications: 48

Step 4: Explanation of the Dynamic Programming Approach

- Initialization:

- dp is a 2D array where dp[i][j] represents the minimum number of scalar multiplications needed to compute the product of matrices from i to j.

- Initially, dp[i][i] is set to 0 because a single matrix doesn't require any multiplications.

- \*Recurrence Relation\*:

- The outer loop (chain\_length) iterates over the possible chain lengths, starting from 2 up to n.

- The inner loops iterate over the possible starting points of the subchains (i) and compute the cost of multiplying matrices within the subchain.

- \*Reconstruction of Optimal Parenthesization\*:

- The optimal\_parenthesization function uses the dp array to construct the optimal parenthesization by recursively selecting the optimal split point.

\*Step 5: Time and Space Complexity Analysis\*

- Time Complexity: O(n^3) where n is the number of matrices. This is due to the triple nested loops used in the dynamic programming approach.

- Space Complexity: O(n^2) for the dp table.

- Efficiency for Large Instances:

- This algorithm is efficient for moderately sized instances (e.g., a few hundred matrices) due to its cubic time complexity. However, for very large instances, more advanced algorithms like Strassen's Algorithm or parallelization techniques may be more suitable.

**ASSIGNMENT 3**

You are tasked with solving the N-Queens problem using a backtracking algorithm. The N-Queens problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. Your goal is to implement the algorithm, find all possible solutions for a given N, and analyze its time complexity.

1. Implement a backtracking algorithm in Python to solve the N-Queens problem.

2. Apply your algorithm to find all possible solutions for a given N (e.g., N = 4 or N = 8). Ensure that you generate all unique solutions.

3. Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).

4. Explain the backtracking approach, including how you generate and validate solutions and how you handle conflicts between queens.

5. Analyze the time complexity of your algorithm and discuss its efficiency for larger values of N.

Input:

• An integer N (N ≥ 4) representing the size of the N×N chessboard and the number of queens to place.

Output:

• All possible solutions to the N-Queens problem for the given N, presented as chessboard representations.

Instructions:

1. Implement a backtracking algorithm to solve the N-Queens problem in Python.

2. Apply your algorithm to find all possible solutions for the provided value of N (e.g., N = 4 or N = 8).

3. Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).

4. Explain the backtracking approach, including solution generation, validation, and conflict resolution.

5. Analyze the time complexity of your algorithm and discuss its performance for larger N values.

Certainly! I'll go through each step as requested:

\*Step 1: Implementing the Backtracking Algorithm\*

python

def solve\_n\_queens(n):

def is\_safe(board, row, col):

for i in range(col):

if board[row][i] == 'Q':

return False

if row - i >= 0 and board[row-i][col-i] == 'Q':

return False

if row + i < n and board[row+i][col-i] == 'Q':

return False

return True

def backtrack(board, col):

nonlocal solutions

if col == n:

solutions.append([''.join(row) for row in board])

return

for i in range(n):

if is\_safe(board, i, col):

board[i][col] = 'Q'

backtrack(board, col + 1)

board[i][col] = '.'

solutions = []

board = [['.' for \_ in range(n)] for \_ in range(n)]

backtrack(board, 0)

return solutions

def print\_chessboard(board):

for row in board:

print(' '.join(row))

\*Step 2 and 3: Applying the Algorithm to Find Solutions for a Given N\*

For example, if N = 4:

python

solutions = solve\_n\_queens(4)

for solution in solutions:

print\_chessboard(solution)

\*Step 4: Explanation of the Backtracking Approach\*

- \*is\_safe Function\*:

- Checks if it's safe to place a queen at a given position (row, col) on the board.

- \*backtrack Function\*:

- Recursively explores the possibilities of placing queens in each column.

- If a solution is found, it's added to the list of solutions.

\*Step 5: Time Complexity Analysis\*

- Time Complexity: O(n!), where n is the size of the chessboard.

- The backtracking algorithm explores all possible combinations of queen placements, leading to a factorial time complexity.

- In practice, this is efficient for small values of n, but can become impractical for larger values.

\*Efficiency for Larger N Values\*:

- For larger N values (e.g., N > 15), the number of possible solutions grows exponentially. This means that finding all solutions becomes increasingly time-consuming. In such cases, more advanced techniques or heuristics might be used to improve performance, or alternative algorithms may be considered.