**ADVANCED DATA STRUCTURES AND ALGORITHMS ASSIGNMENT**

**NAME: D KHOMATHI**

**REGISTER NUMBER: 22CS041**

**DEPARTMENT: COMPUTER SCIENCE AND ENGINEERING**

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **ASSIGNMENT TOPICS** | **PG.NO** |
| **1.** | Sorting a given array using Algorithms  And analysing those algorithms. | **3-6** |
| **2.** | Optimal way to parenthesize the matrix to  Minimize the total number of multiplication  needed to compute the product. | **7-10** |
| **3.** | N-Queens Problem with Backtracking  Algorithm. | **11-14** |

**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 analyse 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. Analyse 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. Analyse 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.

**SOLUTION:**

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]

try:

user\_input = input ("Enter a list of integers separated by spaces: ")

unsorted\_list = list (map (int, user\_input. split()))

except ValueError:

print ("Invalid input. Please enter integers separated by spaces.")

exit ()

bubble\_sort(unsorted\_list)

print ("Sorted array:", unsorted\_list)

**Bubble Sort:**

**Time Complexity:** Bubble Sort has a worst-case time complexity of O(n^2), where n is the number of elements in the array. In the best-case scenario (when the array is already sorted), it has a time complexity of O(n). The average-case time complexity is also O(n^2).

**Stability:** Bubble Sort is a stable sorting algorithm, meaning it maintains the relative order of equal elements.

**Performance on Different Input Data:**

**Best Case:** When the array is already sorted, Bubble Sort performs reasonably well with a time complexity of O(n). However, it still needs to make passes through the array to check if any swaps are needed, so it may not be the most efficient choice for sorted data.

**Worst Case:** When the array is in reverse order, Bubble Sort becomes inefficient with a time complexity of O(n^2). It requires many swaps to bring the largest elements to the end of the array in each pass.

**Average Case:** In most practical cases, Bubble Sort is slower than more efficient sorting algorithms like Quick Sort or Merge Sort. It performs better when the input data is nearly sorted or has a small number of out-of-place elements.

**Comparing Bubble Sort with Quick Sort :**

**Quick Sort:**

**Time Complexity:** Quick Sort has an average-case time complexity of O(n log n), which is significantly better than Bubble Sort's O(n^2). However, it can have a worst-case time complexity of O(n^2) in rare cases.

**Stability: Quick** Sort is not a stable sorting algorithm, so it may not maintain the relative order of equal elements.

**Performance:** Quick Sort is generally preferred over Bubble Sort for larger datasets and general-purpose sorting due to its better average-case time complexity. It efficiently handles a wide range of input data, including both sorted and randomly ordered data.

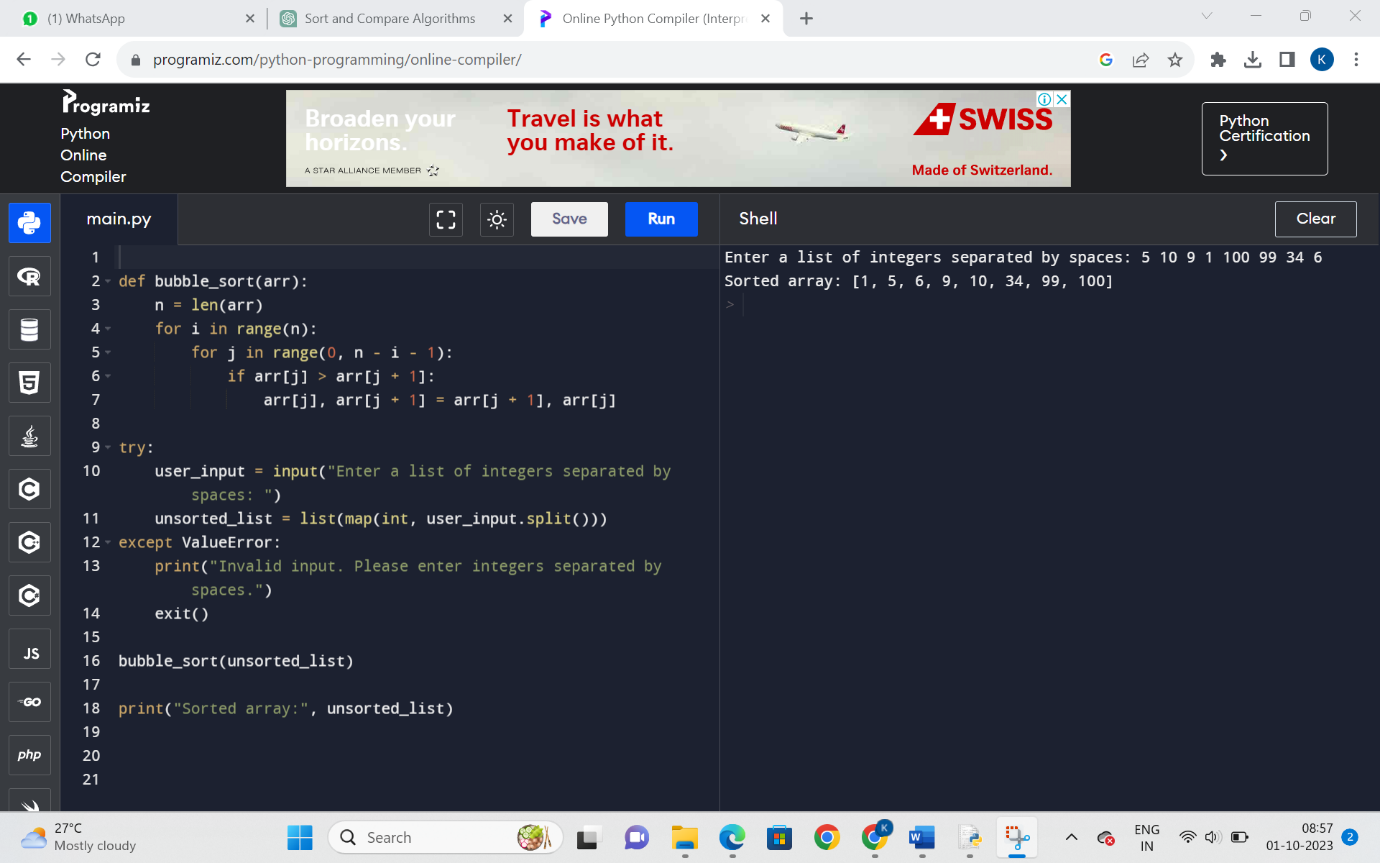
**Scenarios for Choosing Between Bubble Sort and Quick Sort:**

**Bubble Sort:** It may be preferred when dealing with very small arrays or when the data is almost sorted, as it has a lower constant factor for small input sizes and a favorable best-case scenario.

**Quick Sort:** Quick Sort is preferred for larger datasets and when average-case time complexity matters. It is more versatile and efficient for a wide range of input data and is widely used in practice.

In summary, while Bubble Sort is a simple and stable sorting algorithm, it is not efficient for large datasets and is outperformed by Quick Sort in most scenarios. Quick Sort is preferred for general-purpose sorting due to its faster average-case time complexity.

**OUTPUT AFTER IMPLEMENTING IN PYTHON:**

****

**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.

**SOLUTION:**

**Dynamic Programming Algorithm:**

We'll implement the dynamic programming algorithm to find the optimal parenthesization and calculate the minimum number of scalar multiplications. The algorithm involves defining a 2D table to store intermediate results and computing the optimal cost iteratively.

**Python Program:**

def matrix\_chain\_multiplication(matrices):

n = len(matrices)

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

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

for i in range(1, n):

m[i][i] = 0

for l in range(2, n):

for i in range(1, n - l + 1):

j = i + l - 1

m[i][j] = float('inf')

for k in range(i, j):

cost = m[i][k] + m[k + 1][j] + matrices[i - 1][0] \* matrices[k][1] \* matrices[j][1]

if cost < m[i][j]:

m[i][j] = cost

s[i][j] = k

def print\_optimal\_parenthesization(i, j):

if i == j:

print(f'Matrix {i}', end='')

else:

print('(', end='')

print\_optimal\_parenthesization(i, s[i][j])

print\_optimal\_parenthesization(s[i][j] + 1, j)

print(')', end='')

print("Optimal Parenthesization: ", end='')

print\_optimal\_parenthesization(1, n - 1)

print()

return m[1][n - 1]

matrices = [(2, 3), (3, 4), (4, 2)]

min\_scalar\_multiplications = matrix\_chain\_multiplication(matrices)

print("Minimum Scalar Multiplications:", min\_scalar\_multiplications)

**Explanation of Dynamic Programming Approach**

* We use a bottom-up approach with a table **m** to store the minimum number of scalar multiplications and a table **s** to store the optimal split points.
* We initialize the diagonal of **m** to 0 since a single matrix has no multiplications.
* Then, we iterate over chain lengths **l** from 2 to **n**, where **n** is the number of matrices.
* For each chain length and each possible starting matrix **i**, we consider all possible split points **k** between **i** and **j**, calculating the cost of multiplying matrices from **i** to **k** and **k+1** to **j**.
* We update **m[i][j]** with the minimum cost and store the split point **k** in **s[i][j]** if we find a better way to parenthesize.
* Finally, we reconstruct the optimal parenthesization using the **s** table.

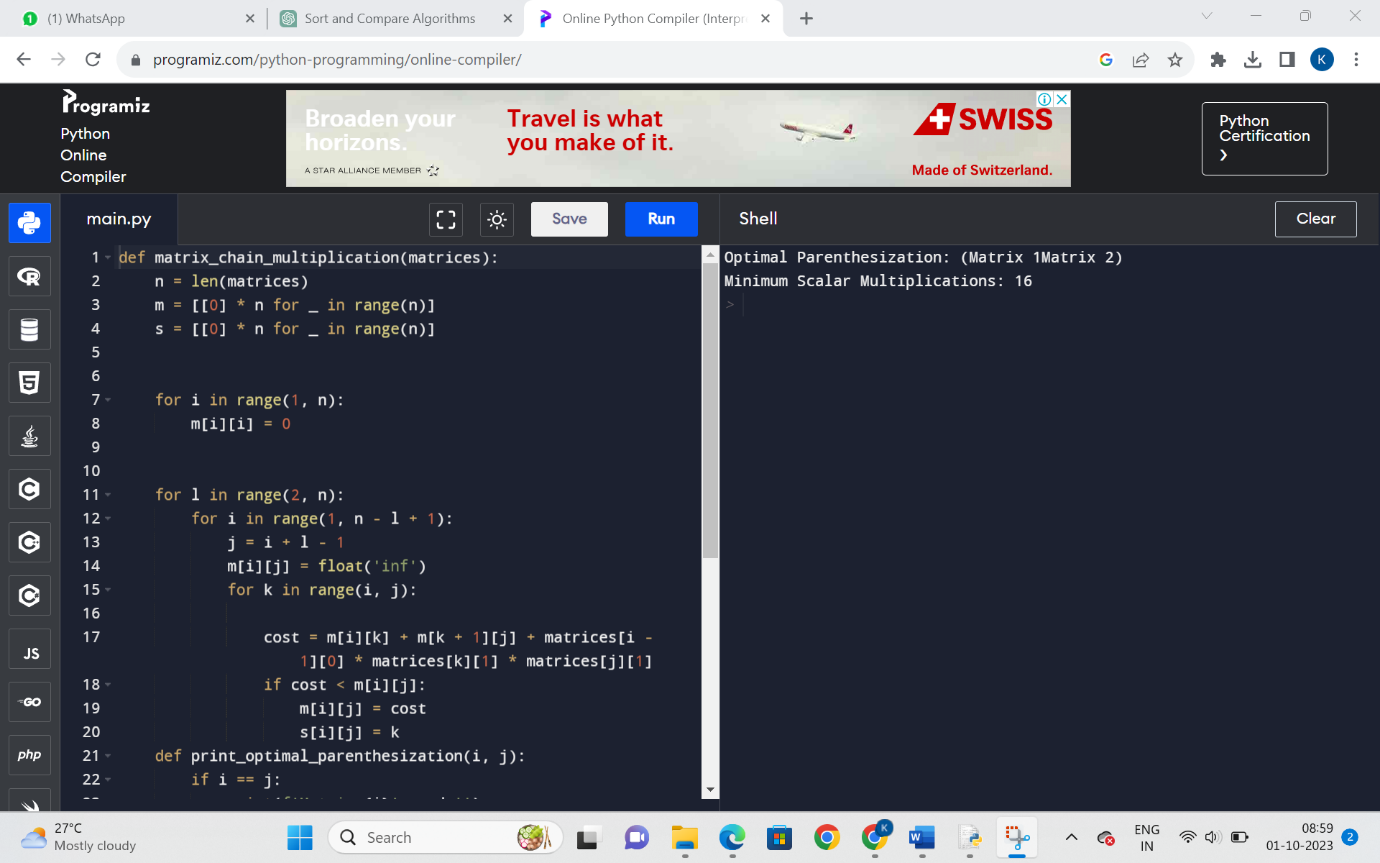
**Time and Space Complexity Analysis**

* **Time Complexity:** The algorithm runs in O(n^3) time, where **n** is the number of matrices. This is because we have three nested loops (chain length, starting matrix, split point).
* **Space Complexity:** The space complexity is O(n^2) to store the **m** and **s** tables.

**Efficiency for Large Instances**

The dynamic programming approach is efficient for solving large instances of the matrix chain multiplication problem because it avoids redundant calculations by storing and reusing intermediate results. However, the time complexity of O(n^3) may become a limiting factor for very large numbers of matrices. In such cases, more advanced algorithms like Strassen's algorithm or parallelization techniques may be considered for further optimization.

**OUTPUT AFTER IMPLEMENTING IN PYTHON:**

****

**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.

**SOLUTION:**

**Implementation of Python Program:**

def solve\_n\_queens(n):

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

for i in range(row):

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

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

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

return False

for i, j in zip(range(row, -1, -1), range(col, n)):

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

return False

return True

def solve(row):

if row == n:

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

return

for col in range(n):

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

board[row][col] = 'Q'

solve(row + 1)

board[row][col] = '.'

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

solutions = []

solve(0)

return solutions

def print\_solutions(solutions):

for i, solution in enumerate(solutions):

print(f"Solution {i + 1}:")

for row in solution:

print(row)

print()

n = 4

solutions = solve\_n\_queens(n)

print\_solutions(solutions)

**Explanation of Backtracking Approach**

* **is\_safe** function checks if it's safe to place a queen in the given position (row, col) on the chessboard by checking if there are no queens in the same column or diagonal.
* **solve** is a recursive function that tries to place queens row by row. When it reaches the last row, it adds a solution to the list of solutions.

**Time Complexity Analysis**

The time complexity of the N-Queens backtracking algorithm is exponential, O(N!), where N is the size of the chessboard. This is because the algorithm explores all possible permutations of queen placements. As N increases, the number of solutions grows rapidly, making the algorithm less efficient for larger values of N.

**Efficiency for Larger N Values**

For small N values (e.g., N = 4 or N = 8), the backtracking algorithm can quickly find and display solutions. However, for larger N values, the number of possible solutions increases exponentially, leading to longer computation times. In practice, for N > 12, the algorithm can become quite slow.

To improve efficiency for larger N values, you may consider optimizations or more advanced techniques like constraint propagation and backjumping. Additionally, parallelizing the search for solutions can help take advantage of multi-core processors and speed up the process.

**OUTPUT AFTER IMPLEMENTING IN PYTHON:**

****