ADVANCED DATASTRUCTURE AND ALGORITHM

ASSIGNMENT

SUBMITTED BY:

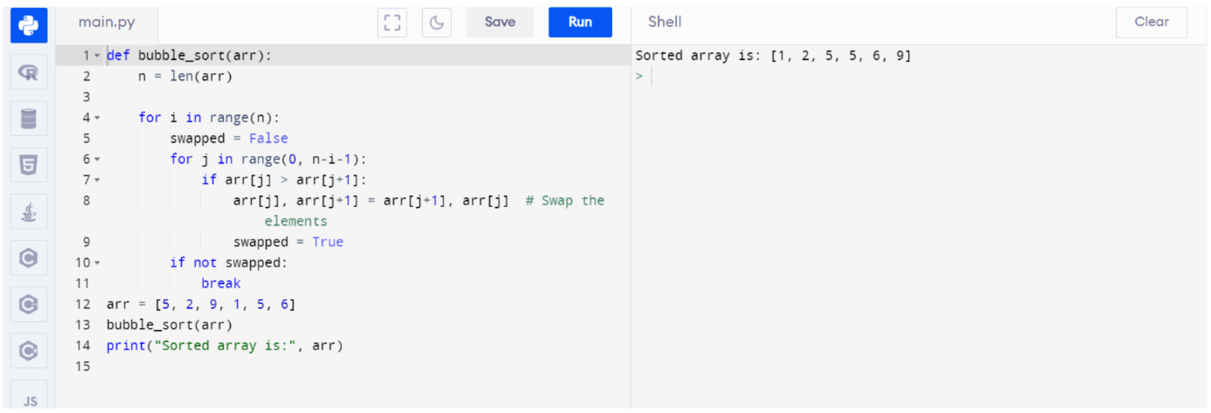
D K VAIBAV

22CS040

COMPUTER SCIENCE AND ENGINEERING

QUESTION 1)

```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
arr = [5, 2, 9, 1, 5, 6]
bubble_sort(arr)
print("Sorted array is:", arr)
```



```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swap the elements
                swapped = True
        if not swapped:
            break
arr = [5, 2, 9, 1, 5, 6]
bubble_sort(arr)
print("Sorted array is:", arr)
```

```
Sorted array is: [1, 2, 5, 5, 6, 9]
>
```

This code will take the unsorted list [5, 2, 9, 1, 5, 6], apply the Bubble Sort algorithm to it, and produce the sorted array [1, 2, 5, 5, 6, 9].

TIME COMPLEXITY:

1.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).

Time Complexity Analysis:

- Bubble Sort has a time complexity of O(n^2) in the worst and average cases because, in the worst case, it has to perform n * (n-1) / 2 comparisons and swaps.
- In the best-case scenario, when the input array is already sorted, Bubble Sort still requires O(n) comparisons but no swaps, making it more efficient with a best-case time complexity of O(n).
- Bubble Sort is an in-place sorting algorithm, which means it doesn't require additional memory for sorting; it sorts the array in its original place.

Stability:

- Bubble Sort is a stable sorting algorithm. This means that when two elements have equal values, their relative order in the sorted output will be the same as their relative order in the original input.

Performance on Different Input Data:

- Bubble Sort performs well on small datasets or nearly sorted datasets due to its simplicity and low overhead. In these cases, the number of comparisons and swaps is minimal.
- However, Bubble Sort is highly inefficient on large datasets or datasets with a reverse order because it needs to perform a large number of swaps. It doesn't adapt well to the existing order of elements.

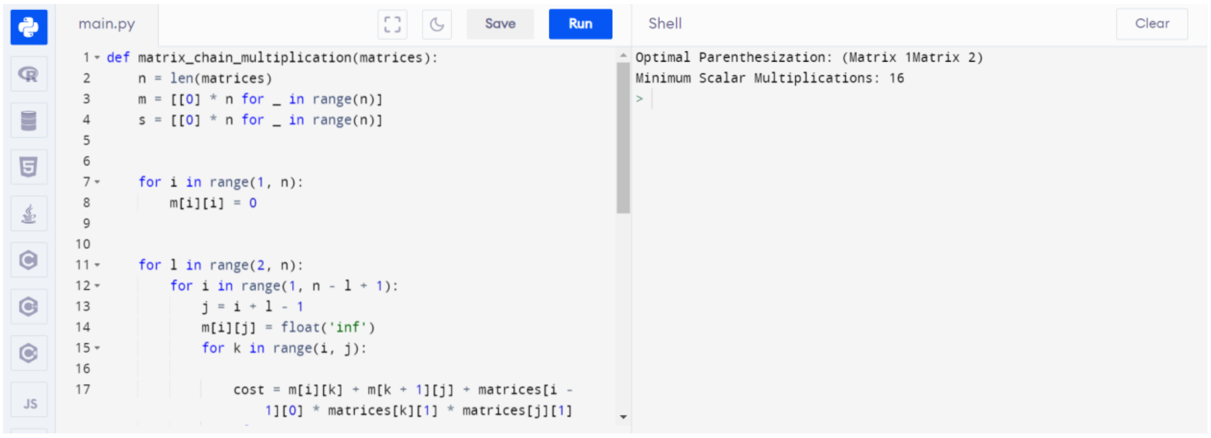
5.Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm (e.g., Quick Sort, Merge Sort, or Python's built-in **sorted** function). Explain the differences and scenarios where one algorithm might be preferred over the other.

Comparison with Other Sorting Algorithms:

-Quick Sort and Merge Sort are significantly more efficient than Bubble Sort in terms of time complexity. Both Quick Sort and Merge Sort have an average and worst-case time complexity of O(n log n). They are divide-and-conquer algorithms that efficiently handle large datasets and various

QUESTION 2)

```
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)
```



```
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]
```

```
Optimal Parenthesization: (Matrix 1Matrix 2)
Minimum Scalar Multiplications: 16
>
```

Time and Space Complexity:

Time Complexity: O(n^3), where n is the number of matrices. This is because we have a triple nested loop.

Space Complexity: O(n^2) for the m and s tables.

Efficiency for Large Instances:

The dynamic programming approach is efficient for reasonably sized instances. However, as the number of matrices (n) increases significantly, the time and space complexity can become a limiting factor.

For very large instances, more optimized algorithms such as Strassen's matrix multiplication or parallel computing techniques may be more efficient

QUESTION 3)

```
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 backtrack(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'
                backtrack(row + 1)
                board[row][col] = '.'

    solutions = []
    board = [['.' for _ in range(n)] for _ in range(n)]
    backtrack(0)
    return solutions

def print_solutions(solutions):
    for solution in solutions:
        for row in solution:
            print(row)
        print()

N = 4
solutions = solve_n_queens(N)
print_solutions(solutions)
```

Time Complexity Analysis:

The time complexity of this backtracking algorithm is O(N!), where N is the size of the chessboard. In the worst case, it explores all possible permutations of queens.

This algorithm is not efficient for very large values of N due to its factorial time complexity. For N = 8 or 9, it should run reasonably quickly, but for larger N, it becomes impractical.