Practical Analysis Report

Here is the detailed analysis of all practicals found in your codebase.

Practical 1.1: Best Chef Competition

File: `Practical1_1.py` 1. **Aim**: Determine the winner between two chefs based on 3 categories. 2. **Problem & Solution**: Compare scores for Presentation, Taste, and Hygiene. The solution iterates through the 3 categories and awards a point to the chef with the higher score in each. 3. **Why it's good**: Simple and fair pairwise comparison method. 4. **Complexity**: **O(1)** (Constant time, as there are always exactly 3 comparisons). 5. **Data Structure**: List.

Practical 1.2: Closest Sum to Zero

File: `Practical1_2.py` 1. **Aim**: Find two elements in an array whose sum is closest to zero. 2. **Problem & Solution**: Given a list of integers, find a pair `(a, b)` such that `|a + b|` is minimized. The solution uses a brute-force approach with nested loops to check every possible pair. 3. **Why it's good**: Guarantees finding the optimal pair by checking all possibilities. 4. **Complexity**: **O(N²)** (Nested loops). 5. **Data Structure**: List.

Practical 2.1: Sum Calculation Analysis

File: `Practical2_1.py` 1. **Aim**: Compare Loop, Equation, and Recursion methods for calculating the sum of N numbers. 2. **Problem & Solution**: Calculate `1 + 2 + ... + N`. * **Loop**: Iterates N times. * **Equation**: Uses `N(N+1)/2`. * **Recursion**: Calls itself N times. 3. **Why it's good**: Demonstrates the efficiency differences between algorithms. The Equation method is instant regardless of N. 4. **Complexity**: * Loop: **O(N)** * Equation: **O(1)** * Recursion: **O(N)** 5. **Graph**: Plots `N` (x-axis) vs `Steps` (y-axis). Shows that Equation is a flat line (constant), while Loop and Recursion are linear. 6. **Data Structure**: List (for plotting data).

Practical 2.2: Fibonacci Sequence Analysis

File: `Practical2_2.py` 1. **Aim**: Compare Iterative and Recursive methods for generating Fibonacci numbers (Rabbit pairs). 2. **Problem & Solution**: Calculate the Nth Fibonacci number. * **Iterative**: Linear loop. * **Recursive**: Tree-like recursive calls. 3. **Why it's good**: Highlights the massive performance cost of naive recursion for this problem. 4. **Complexity**: * Iterative: **O(N)** * Recursive: **O(2■)** (Exponential - very slow for large N). 5. **Graph**: Plots `Months` vs `Steps`. The Recursive line shoots up exponentially, showing it's inefficient compared to Iterative. 6. **Data Structure**: List.

Practical 3.1: Sorting Algorithms Comparison

File: `Practical3_1.py` 1. **Aim**: Compare Bubble Sort, Insertion Sort, and Selection Sort. 2. **Problem & Solution**: Sort an array of random numbers. * **Bubble**: Swaps adjacent elements. * **Insertion**: Builds sorted array one item at a time. * **Selection**: Selects smallest element and swaps. 3. **Why it's good**: Visualizes how step counts grow with input size for quadratic sorting algorithms. 4. **Complexity**: All three are **O(N²)** in this implementation. 5. **Graph**: Plots `Input Size (N)` vs `Steps`. All lines curve upwards (quadratic growth). 6. **Data Structure**: List (Array).

Practical 4.1: Coin Change (Greedy Variant)

File: `Practical4_1.py` 1. **Aim**: Minimize the number of coins needed to make a target sum. 2. **Problem & Solution**: Given denominations, find min coins for a value. The solution sorts coins in descending order and tries to use the largest coins first (Greedy approach). It iterates through different starting coins to find a local minimum. 3. **Why it's good**: Efficient for standard coin systems (like 1, 5, 10, 25). 4. **Complexity**: **O(N × M)** (where N is coin types, M is test cases). 5. **Data Structure**: List.

Practical 5.1 & 8.1: Fractional Knapsack

Files: `Practical5_1.py`, `Practical8_1.py` 1. **Aim**: Maximize profit from items that can be broken into fractions. 2. **Problem & Solution**: You have a knapsack with capacity W. Items have value and weight. You can take fractions of items. Solution: Calculate `Value/Weight` ratio for each item, sort by this ratio (descending), and fill the knapsack. 3. **Why it's good**: The Greedy strategy is

optimal for the *Fractional* Knapsack problem (unlike the 0/1 Knapsack). 4. **Complexity**: **O(N log N)** (dominated by sorting). 5. **Data Structure**: List of Tuples.

Practical 6.1: Matrix Chain Multiplication

File: `Practical6_1.py` 1. **Aim**: Find the most efficient way to multiply a chain of matrices. 2. **Problem & Solution**: Matrix multiplication is associative but the cost depends on the order. The solution uses **Dynamic Programming** to find the order with minimal scalar multiplications. 3. **Why it's good**: Can drastically reduce computation time compared to a naive order. 4. **Complexity**: **$O(N^3)$**. 5. **Data Structure**: 2D Array (DP Table).

Practical 7.1: Longest Common Subsequence (LCS)

File: `Practical7_1.py` 1. **Aim**: Find the longest subsequence present in two sequences. 2. **Problem & Solution**: Given strings P and Q, find the longest common sequence. Solution uses **Dynamic Programming** to build a table of common lengths. 3. **Why it's good**: Fundamental in diff tools, DNA sequencing, and text comparison. 4. **Complexity**: **O(M × N)** (where M and N are lengths of strings). 5. **Data Structure**: 2D Array (DP Table).

Practical 9.1: Huffman Coding

File: `Practical9_1.py` 1. **Aim**: Lossless data compression. 2. **Problem & Solution**: Assign shorter binary codes to more frequent characters. Solution uses a **Min-Heap** to build a Huffman Tree bottom-up. 3. **Why it's good**: Optimal prefix code for data compression. 4. **Complexity**: **O(N log N)**. 5. **Data Structure**: Min-Heap (Priority Queue), Tree.

Practical 10.1: Dijkstra's Algorithm

File: `Practical10_1.py` 1. **Aim**: Find shortest paths from a source node to all other nodes. 2. **Problem & Solution**: Weighted graph with non-negative weights. Solution uses a Greedy strategy: always expand the closest unvisited node. 3. **Why it's good**: Standard algorithm for routing (e.g., GPS, network routing). 4. **Complexity**: **$O(V^2)$** (using adjacency matrix implementation). 5. **Graph Explanation**: The `graph` variable is an **Adjacency Matrix** where `graph[i][j]` represents the weight of the edge from node `i` to `j`. `inf` means no direct edge. 6. **Data Structure**: Adjacency Matrix (2D List).

Practical 11.1: Traveling Salesman Problem (TSP)

File: `Practical11_1.py` 1. **Aim**: Find the shortest route visiting every city exactly once and returning to start. 2. **Problem & Solution**: NP-Hard problem. The solution uses **Brute Force** by generating all permutations of cities to find the minimum cost path. 3. **Why it's good**: Guarantees the exact optimal solution for small N. 4. **Complexity**: **O(N!)** (Factorial - extremely slow for large N). 5. **Graph Explanation**: The `cost` matrix represents the distances between cities. `cost[i][j]` is the distance from city `i` to `j`. 6. **Data Structure**: Adjacency Matrix (2D List).

Potential Evaluation Questions for Practicals

Here are common questions examiners might ask during your practical evaluation, grouped by topic.

General Algorithm Questions

- **What is Time Complexity?** (Big O notation, Best/Average/Worst case)
- **What is Space Complexity?** (Memory usage)

  Difference between Greedy and Dynamic Programming?
  - *Greedy*: Makes the best local choice at each step. Fast but doesn't always guarantee the global optimum.
  - *DP*: Solves all subproblems and stores results to avoid re-computation. Guarantees global optimum but uses more memory.
- **What is Divide and Conquer?** (Break problem into smaller sub-problems, solve them, and combine results).

Practical Specific Questions

1. Sorting (Bubble, Insertion, Selection)

- **Which sorting algorithm is best for small datasets?** (Insertion Sort is often fastest for very small or nearly sorted arrays).
- **Why is Bubble Sort considered inefficient?** (It does many unnecessary swaps).
- **What is the complexity of these sorts?** ($O(N^2)$ for all three in average/worst case).
- **Can you name an O(N log N) sorting algorithm?** (Merge Sort, Quick Sort, Heap Sort).
- **What is a "Stable" sort?** (Preserves the relative order of equal elements).

2. Recursion vs. Iteration (Sum, Fibonacci)

- **Why is the recursive Fibonacci so slow?** (It recalculates the same values many times - overlapping subproblems).
- **What is "Stack Overflow"?** (When recursion goes too deep and runs out of memory).
- **When should you use Recursion?** (When the problem has a recursive structure, like trees or graphs, and code readability is important).

3. Greedy Algorithms (Coin Change, Fractional Knapsack)

- **Does the Greedy Coin Change always work?** (No, it fails for some coin systems, e.g., Coins: [1, 3, 4], Target: 6. Greedy gives 4+1+1 (3 coins), Optimal is 3+3 (2 coins)).
- **Why can we use Greedy for Fractional Knapsack but not 0/1 Knapsack?** (In Fractional, we can always take the "densest" item. In 0/1, taking the densest item might waste space that could be better filled by other items).
- **What is the criteria for sorting in Fractional Knapsack?** (Profit/Weight ratio).

4. Dynamic Programming (LCS, Matrix Chain Multiplication)

- **What is "Memoization"?** (Caching results of function calls).
- **What does `dp[i][j]` represent in LCS?** (Length of LCS between first `i` chars of string A and first `j` chars of string B).
- **Why do we need DP for Matrix Chain Multiplication?** (To find the optimal parenthesization to minimize scalar multiplications. The number of ways to parenthesize is exponential).

5. Huffman Coding

- **What is a Prefix Code?** (No code is a prefix of another. Ensures no ambiguity in decoding).
- **Why do frequent characters get shorter codes?** (To minimize total file size).
- **What data structure is used to build the tree efficiently?** (Min-Heap / Priority Queue).

6. Graph Algorithms (Dijkstra, TSP)

- **What is the difference between Dijkstra and Prim's algorithm?** (Dijkstra finds shortest path from source. Prim's finds Minimum Spanning Tree).
- **Does Dijkstra work with negative weights?** (No, it can get into infinite loops or give wrong answers. Use Bellman-Ford).
- **Why is TSP hard?** (It's NP-Hard. The number of routes grows factorially O(N!), so we can't solve it exactly for large N).
- **What does the adjacency matrix represent?** (Connections and weights between nodes).

Python Specific Questions

- **What is the difference between a List and a Tuple?** (Lists are mutable, Tuples are immutable).
- **What is `float('inf')`?** (Represents infinity, useful for initializing minimum search algorithms).
- **How does `zip()` work?** (Combines multiple iterables element-wise).
- **What is `matplotlib` used for?** (Plotting graphs and visualizations).