🧩 Program 1: Huffman Coding (Greedy Algorithm)

📘 Code Summary:

This program implements **Huffman Coding**, a lossless data compression technique.
It uses a **greedy algorithm** to assign shorter binary codes to more frequent characters.

💬 Viva Questions & Answers (15)

1. **Q1. What is Huffman coding?**
   A: Huffman coding is a data compression algorithm that assigns variable-length binary codes to input characters — shorter codes for more frequent characters.
2. **Q2. Which algorithmic strategy is used in Huffman coding?**
   A: It uses the **Greedy strategy**, selecting the two least frequent nodes repeatedly.
3. **Q3. What is the role of a MinHeap in this code?**
   A: A MinHeap helps efficiently extract the two smallest frequency nodes during each iteration.
4. **Q4. What is the purpose of the MinHeapNode class?**
   A: It represents each character node with its frequency, left child, and right child for tree construction.
5. **Q5. Why do we override the __lt__ method?**
   A: To define custom comparison logic based on frequency, required by the heapq module.
6. **Q6. What does the $ symbol represent in this program?**
   A: It represents an **internal node** in the Huffman tree (not an actual data character).
7. **Q7. How does the algorithm combine nodes?**
   A: It repeatedly removes the two nodes with the smallest frequency, creates a new node with their combined frequency, and re-inserts it into the heap.
8. **Q8. What is the time complexity of Huffman coding?**
   A: O(n log n) due to heap operations.
9. **Q9. What is the space complexity?**
   A: O(n), as it stores all nodes and the Huffman tree.
10. **Q10. What is printed as output?**
    A: Each character and its corresponding Huffman binary code.

11. **Q11. What happens when all characters have equal frequency?**
    A: All will have codes of equal length since no character has higher priority.
12. **Q12. Why is Huffman coding called "lossless"?**
    A: Because no data is lost during compression — the original data can be perfectly reconstructed.
13. **Q13. Where is Huffman coding used in real life?**
    A: In JPEG image compression, ZIP file compression, and MP3 audio encoding.
14. **Q14. What Python module is used for heap implementation?**
    A: The built-in heapq module.
15. **Q15. What happens in the print_codes function?**
    A: It traverses the Huffman tree recursively to print binary codes for each character.
16. • **Q16. Why do we use a priority queue (MinHeap) instead of a normal list?**
    A: Because MinHeap gives efficient access to the smallest frequency element in **O(log n)** time.
17. • **Q17. What happens if there's only one character in the input?**
    A: The algorithm assigns a single code, usually "0", since there's no need for a tree.
18. • **Q18. How is Huffman tree different from a Binary Search Tree (BST)?**
    A: A Huffman tree is built based on frequency, not key order. It's a **full binary tree**, not ordered like BST.
19. • **Q19. Why are Huffman codes prefix-free?**
    A: No code is a prefix of another, ensuring unambiguous decoding.
20. • **Q20. What is the output if all characters have same frequency?**
    A: Each character gets a code of the same length, since priority doesn't change.
21. • **Q21. How do you calculate total bits saved after compression?**
    A: Compare total bits before and after using Huffman codes based on character frequencies.
22. • **Q22. Why do we use recursion in `print_codes()`?**
    A: To traverse the Huffman tree using depth-first traversal and generate codes for each leaf node.
23. • **Q23. What type of tree is generated by Huffman coding?**
    A: A **binary tree** where each leaf node represents a character.

24. • **Q24. Can Huffman coding handle non-text data (like images)?**

    A: Yes, any data with frequency distribution (e.g., pixel values) can be compressed.

25. • **Q25. What is the limitation of Huffman coding?**

    A: It doesn't adapt dynamically to data — static Huffman coding requires prior frequency knowledge.

👑 **Program 2: N-Queens Problem (Backtracking)**

📘 **Code Summary:**

This program solves the **N-Queens problem** using **backtracking**, with a twist — the **first queen's column position is fixed** by the user.

💬 **Viva Questions & Answers (15)**

1. **Q1. What is the N-Queens problem?**
   A: Placing N queens on an N×N chessboard so that no two queens attack each other.
2. **Q2. What technique is used to solve it?**
   A: Backtracking, a recursive algorithm to explore all possible placements.
3. **Q3. How is safety of a position checked?**
   A: By ensuring the same column and both diagonals have no other queen.
4. **Q4. What are col, posDiag, and negDiag used for?**
   A: They store occupied columns, positive diagonals (r+c), and negative diagonals (r−c) respectively.
5. **Q5. What happens when a valid configuration is found?**
   A: The board state is added to the res list as a valid solution.
6. **Q6. What does backtrack(r) do?**
   A: It attempts to place a queen in row r by testing valid columns recursively.
7. **Q7. What is the base case in recursion?**
   A: When r == n, meaning all queens are successfully placed.

8. **Q8. What is meant by "backtracking"?**
   A: Reverting a previous step (removing a queen) when no valid move exists further.
9. **Q9. How does fixing the first queen affect solutions?**
   A: It reduces the total number of configurations by fixing one queen's column.
10. **Q10. Why are we using sets instead of lists?**
    A: Sets provide **O(1)** lookup for checking occupied positions.
11. **Q11. What is printed as output?**
    A: A board matrix of 1s (queen) and 0s (empty spaces).
12. **Q12. What is the time complexity?**
    A: O(N!) in the worst case.
13. **Q13. What is the space complexity?**
    A: $O(N^2)$ for the board and recursive stack.
14. **Q14. Can this algorithm find all possible solutions?**
    A: Yes, by continuing recursion even after finding one valid configuration.
15. **Q15. In real life, where is backtracking used?**
    A: In puzzles like Sudoku, crosswords, pathfinding, and constraint satisfaction problems.
16. • **Q16. Why can't we place two queens in the same column?**
    A: Because queens attack vertically, and each column can only have one queen.
17. • **Q17. Why do we represent diagonals as (r + c) and (r - c)?**
    A: It uniquely identifies each diagonal — all cells with the same (r + c) or (r - c) are on the same diagonal.
18. • **Q18. What is the advantage of using sets for column and diagonal tracking?**
    A: They allow constant-time lookup (**O(1)**), improving efficiency.
19. • **Q19. What does backtracking mean in a single sentence?**
    A: Trying all possibilities by placing a queen and undoing if it leads to no solution.
20. • **Q20. What would happen if we remove the backtrack step (removal code)?**
    A: The algorithm would fail because it wouldn't explore other configurations.
21. • **Q21. Why do we start recursion from row 1 instead of 0 here?**
    A: Because the first queen's column is fixed by the user, so row 0 is already filled.

22. • **Q22. Can we solve N-Queens iteratively?**

    A: It's possible but very complex. Recursive backtracking is more natural for this problem.

23. • **Q23. What is the output format of this program?**

    A: It prints the chessboard as a matrix with "1" for queen positions and "0" for empty squares.

24. • **Q24. What is the minimum N for which the N-Queens problem has a solution?**

    A: For $N \geq 4$, the problem has valid solutions.

25. • **Q25. What is the significance of fixing the first queen's column?**
    A: It helps explore reduced configurations or test specific placements for optimization.

🧠 **Program 3: Fibonacci Series — Recursive vs Iterative Analysis**

📘 **Code Summary:**

This code compares **recursive** and **iterative** Fibonacci implementations, measuring **execution time** and **memory usage** using time and tracemalloc.

💬 **Viva Questions & Answers (15)**

1. **Q1. What is the Fibonacci series?**
   A: A sequence where each term is the sum of the two preceding ones: 0, 1, 1, 2, 3, 5, 8...
2. **Q2. How is Fibonacci implemented recursively?**
   A: Each call computes F(n-1) + F(n-2) until n <= 1.
3. **Q3. Why is recursion inefficient for Fibonacci?**
   A: Because it recomputes the same values multiple times (exponential complexity).
4. **Q4. What is the iterative approach?**
   A: Using a loop to compute Fibonacci numbers sequentially — efficient and faster.
5. **Q5. What is the time complexity of recursive Fibonacci?**
   A: $O(2^n)$ due to repeated computations.

6. **Q6. What is the time complexity of iterative Fibonacci?**
   A: O(n) — linear time.
7. **Q7. What is the space complexity of recursive Fibonacci?**
   A: O(n) due to function call stack.
8. **Q8. What is the space complexity of iterative Fibonacci?**
   A: O(1) — uses only two variables.
9. **Q9. Why is tracemalloc used?**
   A: To measure the current and peak memory usage during function execution.
10. **Q10. What is the purpose of gc.collect()?**
    A: It forces garbage collection to clear unused memory before measurement.
11. **Q11. What is time.perf_counter() used for?**
    A: It provides high-resolution timing to measure execution duration accurately.
12. **Q12. Why do we compare both methods?**
    A: To demonstrate performance trade-offs between recursion and iteration.
13. **Q13. What happens if n is large (e.g., 40 or 50)?**
    A: Recursive function becomes extremely slow due to exponential growth of calls.
14. **Q14. Which approach is better for large n?**
    A: The **iterative** approach — faster and memory-efficient.
15. **Q15. What real-world concept does this program teach?**
    A: It shows how **algorithm design choices** (recursion vs iteration) affect **time and space efficiency**.
16. **Q16. Why is recursion slower for Fibonacci?**
    A: It repeats many subproblems without storing results — leading to **exponential growth** in calls.
17. **Q17. How can we optimize recursive Fibonacci?**
    A: By using **Dynamic Programming (Memoization)** to store computed results.
18. **Q18. What is memoization?**
    A: A technique to cache results of previous computations to avoid redundant recursion.
19. **Q19. What is the output for n=0 and n=1?**
    A: For n=0 → 0, and for n=1 → 1 (base cases).

20. **Q20. Why is iterative Fibonacci more memory efficient?**

    A: It uses only two variables (`a`, `b`) and no function call stack.

21. **Q21. What is the use of the `tracemalloc` module?**

    A: It tracks memory allocation to measure how much memory the function uses.

22. **Q22. Why is `time.perf_counter()` preferred over `time.time()`?**

    A: It provides **higher precision** and includes time spent in sleep and OS calls.

23. **Q23. What's the main difference in time complexity between recursive and iterative Fibonacci?**

    A: Recursive is **O(2^n)**; Iterative is **O(n)**.

24. **Q24. What happens if we increase n to 40 or more in recursive Fibonacci?**

    A: The program becomes extremely slow and consumes a lot of memory.

25. **Q25. Why do we compare both methods in one program?**

    A: To understand the **impact of algorithm design** on performance — a key learning goal in Data Structures & Algorithms.

Practical 4: Fractional Knapsack Problem (Greedy Algorithm)

💡 **Concept Summary:**

- You're given items with **weights and values**.
- The goal: **maximize total value** in the knapsack with limited capacity.
- Unlike 0/1 knapsack, you can take **fractions** of items.
- Sorting is done by **value-to-weight ratio (v/w)**.

🧠 **Top 25 Viva Questions & Answers**

1. **Q1. What is the Fractional Knapsack problem?**
   A: It's an optimization problem where items can be broken into fractions to maximize total profit within a limited capacity.

2. **Q2. What strategy is used to solve this problem?**
   A: The **Greedy Strategy** — always take the item with the highest value-to-weight ratio first.

3. **Q3. What is the formula for the ratio?**
   A: Value/Weight (v/w).

4. **Q4. Why is it called "fractional"?**
   A: Because we can take a portion (fraction) of an item instead of taking it whole.

5. **Q5. What's the difference between 0/1 and fractional knapsack?**
   A: In 0/1, you can't split items (take or leave), while in fractional, you can take parts.

6. **Q6. What is the time complexity?**
   A: O(n log n), due to sorting by ratio.

7. **Q7. Why do we sort the items?**
   A: To ensure we pick the most valuable items first based on ratio.

8. **Q8. What data structure is best for this?**
   A: A list or array, often sorted using Python's sorted().

9. **Q9. Can fractional knapsack be solved using dynamic programming?**
   A: No, it doesn't require DP; greedy gives the optimal solution.

10. **Q10. What happens if two items have the same ratio?**
    A: Any order gives the same result, as both contribute equally per weight.

11. **Q11. What is the base condition for capacity?**
    A: Stop when total capacity becomes 0.

12. **Q12. How do you handle partial items in code?**
    A: Take a fraction: total_value += remaining_capacity * (value / weight).

13. **Q13. Why do we break after partial item selection?**
    A: Because the knapsack is full.

14. **Q14. What is the best case scenario?**
    A: When all items fit entirely within capacity.

15. **Q15. What is the worst case scenario?**
    A: When only one item can be partially taken.

16. **Q16. What is greedy choice property?**
    A: Making the best local choice (max ratio) leads to the global optimal solution.

17. **Q17. What is optimal substructure?**
    A: The optimal solution can be built from optimal solutions of smaller subproblems.
18. **Q18. What kind of problem is it (min/max)?**
    A: A **maximization problem**.
19. **Q19. What is the output of the algorithm?**
    A: Maximum possible value for the given capacity.
20. **Q20. How do you verify if your solution is correct?**
    A: By manually calculating ratios and simulating the selection process.
21. **Q21. What happens if weights are 0?**
    A: It's invalid since ratio (v/w) becomes infinite.
22. **Q22. How to handle user input for items?**
    A: By taking pairs of values and weights from input.
23. **Q23. What Python function helps in sorting by ratio?**
    A: sorted(items, key=lambda x: x[0]/x[1], reverse=True)
24. **Q24. What is the final output printed in your program?**
    A: "Maximum value by value/weight ratio: <value>"
25. **Q25. Real-life example?**
    A: Loading a truck with goods of varying weights and profits to maximize revenue.

💻 **Practical 5: 0/1 Knapsack Problem (Dynamic Programming)**

💡 **Concept Summary:**

- You have items with **values** and **weights**.
- You must **choose items** to maximize total value without exceeding capacity.
- You can't take fractions — either include or exclude.
- Uses **Dynamic Programming** (DP) to build a 2D table.

🧠 **Top 25 Viva Questions & Answers**

1. **Q1. What is the 0/1 Knapsack problem?**
   A: A problem where we must select items to maximize total value without exceeding capacity — either include (1) or exclude (0) each item.

2. **Q2. What technique is used here?**
   A: Dynamic Programming (DP).

3. **Q3. Why not Greedy?**
   A: Greedy doesn't work here because items can't be fractionally divided.

4. **Q4. What is the recursive formula?**
   A:
   dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
   if weight fits.

5. **Q5. What is the base case?**
   A: When i=0 or w=0, dp[i][w]=0.

6. **Q6. What does dp[i][w] represent?**
   A: Maximum value attainable using first i items with capacity w.

7. **Q7. What is the size of the DP table?**
   A: (n+1) × (W+1).

8. **Q8. What is the time complexity?**
   A: O(n × W).

9. **Q9. What is the space complexity?**
   A: O(n × W) (can be reduced to O(W)).

10. **Q10. What happens if item's weight > capacity?**
    A: We skip the item: dp[i][w] = dp[i-1][w].

11. **Q11. How is the table filled?**
    A: Row by row, where each cell depends on previous computations.

12. **Q12. What does the last cell dp[n][W] represent?**
    A: The **maximum profit** obtainable.

13. **Q13. What is the method used to find which items were selected?**
    A: Backtracking from dp[n][W].

14. **Q14. What condition tells us an item was included?**
    A: If dp[i][w] != dp[i-1][w], item i-1 was included.

15. **Q15. How do we update capacity during backtracking?**
    A: Subtract the included item's weight from w.

16. **Q16. Why is DP better than recursion here?**
    A: DP avoids recomputation by storing results (overlapping subproblems).

17. **Q17. What kind of problems use DP?**
    A: Problems with **overlapping subproblems** and **optimal substructure**.

18. **Q18. What does "optimal substructure" mean?**
    A: The solution to a problem can be built from its subproblem solutions.

19. **Q19. What happens if all weights are larger than capacity?**
    A: Maximum value = 0.

20. **Q20. What does selected_items array store?**
    A: The indices of items included in the optimal solution.

21. **Q21. Can we solve it recursively?**
    A: Yes, but it's inefficient without memoization.

22. **Q22. What real-world applications use 0/1 knapsack?**
    A: Budget optimization, resource allocation, cargo loading.

23. **Q23. Why do we initialize DP table with zeros?**
    A: It represents no items or zero capacity → zero profit.

24. **Q24. Can we use floating weights or values?**
    A: Usually integers; fractional weights fit fractional knapsack, not 0/1.

25. **Q25. What's printed in the output?**
    A: "Maximum value: <value>" and "Selected items: <indices>".

No.	Practical Title / Algorithm	Key Idea / Approach	Working Summary	Time Complexity	Applications / Use Cases	Key Viva Questions
1	**DFS (Depth First Search)**	Uses **stack / recursion** to explore as deep as possible before backtracking.	Start from node → explore one branch fully → backtrack → explore next branch.	O(V + E)	Pathfinding, maze solving, topological sort	1. What data structure is used in DFS?2. How DFS differs from BFS?3. Applications of DFS?4. Can DFS detect cycles?5. Space complexity?
2	**BFS (Breadth First Search)**	Uses **queue** to explore nodes level by level.	Start from node → explore all neighbors → then next level nodes.	O(V + E)	Shortest path in unweighted graph, social network search	1. What data structure is used in BFS?2. Where BFS is preferred over DFS?3. BFS real-life example?4. BFS vs DFS difference?5. BFS output order example?
3	*A Algorithm (Informed Search)**	Uses **heuristics (h(n)) + cost (g(n))** to find optimal path.	f(n) = g(n) + h(n); chooses path with lowest f(n).	O(E) (depends on heuristic)	Route navigation (Google Maps), games	1. What is f(n)?2. What is heuristic?3. A* vs Dijkstra?4. What is admissible heuristic?5. A* application?
4	**N-Queens Problem (Backtracking)**	Place **N queens** on N×N board so	Try placing queen row by row → check column &	O(N!)	Chess puzzle, constraint satisfaction	1. What is backtracking?2. How many diagonals checked?3. Base

No.	Practical Title / Algorithm	Key Idea / Approach	Working Summary	Time Complexity	Applications / Use Cases	Key Viva Questions
		no two attack each other.	diagonals → backtrack if conflict.			condition of recursion?4. Meaning of safe position?5. Time complexity?
5	**Fibonacci (Recursive vs Iterative)**	Compare **recursion** vs **iteration** in generating Fibonacci sequence.	Recursive calls → stack overhead; iterative → simple loop.	Recursive: $O(2^n)$Iterative: O(n)	Dynamic programming, financial modeling	1. Why recursion is slower?2. Which uses more memory?3. Define base case?4. Example of recursion in AI?5. Output for n=5?