Q: Recursion

1. Problem Understanding

- Recursion is when a function calls itself directly or indirectly to solve a smaller instance of the same problem.
- Goal:
- Break a big problem into smaller subproblems until reaching a base case, then combine results on the way back.
- **Essential Components of Recursion**
  - Base Case (Stopping Condition):
    - Prevents infinite recursion.
    - Represents the simplest subproblem whose answer is known directly.
    - Example:
    - if (n == 0) return;
  - Recursive Case:
    - The function calls itself on smaller input.
    - Example:
    - recursion(n - 1);
  - Work Before / After Recursive Call:
    - Decides output order (e.g., pre-order, post-order).
    - Example:
    - System.out.println(n); // before → descending
    - recursion(n - 1);
    - System.out.println(n); // after → ascending
- **How Recursion Works Internally**
  - Every function call is pushed onto the call stack.
  - When the base case is reached, functions start returning one by one (stack unwinds).
  - Think of recursion as stack behavior (LIFO – Last In, First Out).
- **Types of Recursion**
  - Direct Recursion
    - Function calls itself directly.
    - void fun() { fun(); }
  - Indirect Recursion
    - Function A calls B, and B calls A.
    - void A() { B(); }
    - void B() { A(); }
  - Tail Recursion
    - The recursive call is the last statement in the function.

        - No computation after the recursive call.
        - Can be optimized to iteration.
        - void print(int n) {
            - if (n == 0) return;
            - System.out.println(n);
            - print(n - 1); // tail call
        - }
    - Non-Tail Recursion
        - Some work remains after the recursive call returns.
        - void print(int n) {
            - if (n == 0) return;
            - print(n - 1);
            - System.out.println(n);
        - }
    - Multiple Recursion
        - More than one recursive call inside the same function.
        - int fib(int n) {
            - if (n <= 1) return n;
            - return fib(n - 1) + fib(n - 2);
        - }

- **Important Recursion Concepts**

    - Recursive Tree
        - Used to visualize how recursion unfolds.
        - Helps understand time complexity.
        - Each recursive call forms a branch in the tree.
    - Recurrence Relation
        - Mathematical equation that defines the runtime of recursion.
        - Example:
        - For factorial → $T(n) = T(n-1) + O(1)$
        - For merge sort → $T(n) = 2T(n/2) + O(n)$
    - Master Theorem (for Divide and Conquer)
        - Used to find complexity of recursive relations like:
        - $T(n) = aT(n/b) + f(n)$
        - Then:
        - If $f(n) = O(n^{(\log_b a - \varepsilon)}) \rightarrow T(n) = \Theta(n^{(\log_b a)})$
        - If $f(n) = \Theta(n^{(\log_b a)}) \rightarrow T(n) = \Theta(n^{(\log_b a)} \log n)$
        - If $f(n) = \Omega(n^{(\log_b a + \varepsilon)}) \rightarrow T(n) = \Theta(f(n))$
        - Example:
        - Merge Sort → $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

- **Base Case Design Tips**

    - Always handle smallest input first (n=0, empty array, single node, etc.)
    - Prevent calls like recursion(-1) or infinite loops.
    - For counting/returning recursion, base case usually returns a constant (0 or 1).

- **Tricks for Recursion**
  - Break problems into “smaller same-type subproblems”.
  - Always define what your function means — “What does recursion(n) represent?”
  - Work backwards from base case to build logic.
  - Use recursion when:
    - You can define the solution in terms of smaller subproblems.
    - Problem naturally follows divide & conquer or backtracking.
  - Convert recursion → iteration for optimization if required.
- **Common Mistakes**
  - Missing base case (infinite recursion).
  - Wrong return statement (losing result of recursive call).
  - Forgetting to reduce input (no progress toward base case).
  - Confusing pre/post recursion output.
  - Ignoring stack overflow for large inputs.
- **Recursion vs Iteration**
  - Recursion: Uses function call stack; may lead to stack overflow if not optimized.
  - Iteration: Uses loops and variables; usually more space-efficient.
- **Time & Space Complexity**
  - Time Complexity: Number of recursive calls × time per call.
  - Space Complexity: Due to the recursion call stack (O(depth of recursion)).
- **Tips & Tricks**
  - Always define a clear base case.
  - Reduce problem size in each recursive step.
  - Visualize recursion using the call stack.
  - Use memoization to optimize repeated calls.
  - Tail recursion can be optimized to avoid stack overflow.

Q3: Recursion with Backtracking

1. Problem Understanding

- **Recursion**
  - A function that calls itself to solve smaller subproblems.
  - Components:
    - Base case: Stops recursion.
    - Recursive case: Calls function with smaller input.
  - Usually computes one solution.
  - Examples: Factorial, Fibonacci, GCD.

- **Backtracking**
    - A special type of recursion for exploring all possible solutions.
    - Follows "try → check → undo" pattern.
    - Steps:
        - Choose a possibility.
        - Explore recursively.
        - Check if it leads to a valid solution.
        - Undo / Backtrack to try other possibilities.
    - Examples: N-Queens, Sudoku, Maze solving, Permutations/Combinations.

- **Key Differences between Recursion and Backtracking**
    - Recursion solves a problem; backtracking explores all solutions.
    - Recursion doesn't need to undo choices; backtracking does.
    - Recursion often finds one solution, backtracking finds all valid solutions.

- **Characteristics of Backtracking**
    - Decision-making at each step.
    - Recursive exploration of choices.
    - Undoing invalid choices to explore alternatives.
    - Prunes invalid paths to save computation.

- **Backtracking Template:**
    - void backtrack(parameters) {
        - if (base_case_condition) {
            - // process solution
            - return;
        - }
        - for (each choice) {
            - if (choice is valid) {
                - make choice;
                - backtrack(next_state);
                - undo choice; // backtrack
            - }
        - }
    - }

- **When to Use Backtracking**
    - Combinatorial problems: subsets, permutations, combinations.
    - Constraint satisfaction problems: Sudoku, N-Queens, crossword puzzles.
    - Path-finding problems: Maze solving, word search in a grid.

- **Important Notes:**
    - All backtracking problems are recursive, but not all recursion is backtracking.
    - Backtracking systematically explores all paths and prunes invalid branches.
