Section A: Subqueries (Scalar, Row, Column, IN, EXISTS, ANY, ALL)

1. Find the customers who have **never made a purchase** using `NOT IN` subquery.
2. Display customer names who have made **purchases greater than the average sale amount**.
3. Get the **total sales** for each customer using a correlated subquery.
4. List customers who made **at least one purchase in 2022** using `EXISTS`.
5. Retrieve customers who purchased **every available product** (using `ALL`).
6. Show customers who have **more total purchases than customer ID 10** (using `ANY`).
7. Get the name of the **top-spending customer** (using scalar subquery + `LIMIT`).
8. Find customers from a city where **the average sale amount is above 600**.
9. Show products that were **never purchased by customers in Chicago**.
10. Display customers whose **sales exceeded their own average sale** (correlated subquery).

Section B: Views (Create, Replace, Drop, Updatable vs Read-Only)

11. Create a view to show only `customer_id`, `customer_name`, and `city`.
12. Create a view `high_spenders` showing customers with total sales > 5000.
13. Replace the `high_spenders` view to now include city as well.
14. Drop the `high_spenders` view.
15. Create a view to show all sales made in 2023 with product and sale amount.
16. Create a read-only view using an aggregate function and explain why it's not updatable.
17. Write a query using a view to show customer names and the count of purchases.
18. Create a view that **hides customer join date** from analysts.
19. Create a view for city-wise total sales, grouped by city.
20. Demonstrate a role-based view: customers only from `'Los Angeles'`.

Section C: Indexes (Create, Drop, Composite, Performance Impact)

21. Create an index on the `city` column of the `customers` table.
22. Create a composite index on `(customer_id, product)` in the `sales` table.
23. Drop the index on the `city` column.
24. Explain what happens when you insert 10,000 rows into a table with 5 indexes.
25. Create an index on `sale_date` and run a query to see performance difference with `EXPLAIN`.
26. Show an example where indexing a **low-cardinality column** (like gender) doesn't help.

27. Write a query where composite index on `(customer_id, product)` helps.
28. Write a query where that same index doesn't help (e.g., only filtering by `product`).
29. Create a query that benefits from both a `WHERE` and an `ORDER BY` index.
30. Explain how too many indexes can slow down `UPDATE` or `DELETE` operations with examples.

Section D: Partitioning (Range, How to View, Insert, Query)

31. Create a `sales_partitioned` table partitioned by **YEAR(sale_date)** (range).
32. Create at least **4 partitions** for years 2020–2023.
33. Insert values into each partition manually and confirm they land in the correct one.
34. Query `INFORMATION_SCHEMA.PARTITIONS` to list all partitions of the `sales_partitioned` table.
35. Write a query to get **total sales in 2021** from partitioned table.
36. Add a catch-all `pmax` partition for future sales.
37. Explain how the **PRIMARY KEY must include partition key** in MySQL.
38. Compare query performance for year-based filtering with and without partition.
39. Show an example of how `DROP PARTITION` can be used to **archive old data**.
40. Create a monthly partition for `sale_date` and verify inserts across months.

Section E: Integration + Higher-Order Thinking

41. Create a view that **combines partitioned sales data** and customer names via a `JOIN`.
42. Write a query using a **subquery inside a view** to show top 5 customers by sales.
43. Run `EXPLAIN` on a query with both a **view and an index** and analyze its efficiency.
44. Demonstrate how **index + partition + view** can be combined for a dashboard query.
45. Create a report query that uses: a view, a subquery, a join, and a filtered index (advanced full-scope test).