Of course. Here is a comprehensive list of answers to the SQL interview questions, formatted for clarity and interview-readiness.

Basic SQL Questions (23 Questions)

Database Fundamentals

Question 1: What is SQL?

Theory
SQL, which stands for **Structured Query Language**, is a standardized programming language specifically designed for managing and manipulating data stored in a relational database management system (RDBMS). It is used to perform tasks such as querying data, inserting, updating, and deleting records, and managing the database schema.

Question 2: What is a database?

Theory
A database is an organized collection of structured information, or data, typically stored electronically in a computer system. It is designed to be easily accessed, managed, and updated. Most databases use a structured query language (SQL) for writing and querying data.

Question 3: What are the main types of SQL commands?

Theory
SQL commands are divided into several main categories based on their functionality. These are often referred to as sublanguages of SQL. The main types are:

1. **DDL (Data Definition Language)**: Defines the database schema.
2. **DML (Data Manipulation Language)**: Used for adding, deleting, and modifying data.
3. **DCL (Data Control Language)**: Manages access rights and permissions.
4. **TCL (Transaction Control Language)**: Manages transactions in the database.
5. **DQL (Data Query Language)**: Used to query the data (though often considered a part of DML).

Question 4: What is DDL, DML, DCL, and TCL?

Theory

- **DDL (Data Definition Language)**: These commands are used to define and manage the structure of database objects like tables, indexes, and views.
    - **Commands**: CREATE, ALTER, DROP, TRUNCATE, RENAME.
-
- **DML (Data Manipulation Language)**: These commands are used to manipulate the data within the tables.
    - **Commands**: SELECT, INSERT, UPDATE, DELETE.
-
- **DCL (Data Control Language)**: These commands are used to control access to data in the database, managing user permissions.
    - **Commands**: GRANT, REVOKE.
-
- **TCL (Transaction Control Language)**: These commands are used to manage transactions, which are sequences of operations performed as a single logical unit of work.
    - **Commands**: COMMIT, ROLLBACK, SAVEPOINT.
-

Question 5: What is a primary key?

Theory
A **primary key** is a constraint that uniquely identifies each record in a table.
Characteristics:

1. **Uniqueness**: A primary key must contain unique values for each row.
2. **Non-Null**: A primary key column cannot have NULL values.
3. **Single Key**: A table can have only one primary key. This key can consist of a single column (a simple key) or multiple columns (a composite key).

Question 6: What is a foreign key?

Theory
A **foreign key** is a key used to link two tables together. It is a field (or collection of fields) in one table that refers to the **primary key** in another table. The table containing the foreign key is called the child table, and the table containing the primary key is called the parent table. This constraint is used to enforce **referential integrity**.

Question 7: What is the difference between SQL and NoSQL databases?

Theory

Feature	SQL (Relational Databases - RDBMS)	NoSQL (Non-relational Databases)
Data Model	Structured, with a predefined schema (tables, rows, columns).	Dynamic schema or schema-less (documents, key-value, graph, wide-column).
Schema	**Schema on write**: The schema must be defined before data is written.	**Schema on read**: The schema is flexible and can be defined at read time.
Scalability	Primarily **vertically scalable** (increase the power of a single server).	Primarily **horizontally scalable** (distribute the load across multiple servers).
Consistency	Strong consistency (**ACID** properties).	Tunable consistency, often favoring **BASE** (Basically Available, Soft state, Eventually consistent).
Query Language	Uses SQL.	Varies by database (e.g., MongoDB uses a JSON-based query language).
Examples	MySQL, PostgreSQL, SQL Server, Oracle.	MongoDB, Cassandra, Redis, Neo4j.

Question 8: What are the different data types in SQL?

Theory

SQL supports a wide range of data types to store different kinds of information. The exact names and syntax can vary slightly between different RDBMSs, but the main categories are:

1. **String Types**: CHAR, VARCHAR, TEXT.
2. **Numeric Types**: INT (or INTEGER), SMALLINT, BIGINT, DECIMAL, NUMERIC, FLOAT, REAL.
3. **Date and Time Types**: DATE, TIME, DATETIME, TIMESTAMP.
4. **Binary Types**: BINARY, VARBINARY, BLOB.
5. **Boolean Types**: BOOLEAN (or often BIT or TINYINT in some systems).

Question 9: What is a table and a field in SQL?

Theory

- **Table**: A table is the primary database object that stores data in a structured format of rows and columns. It is a collection of related data entries.
- **Field**: A field, also known as a **column**, is a vertical entity in a table that contains all the information of a specific type for all the records. Each field has a data type.

Question 10: What are constraints in SQL?

Theory
Constraints are rules that are applied to the columns of a table to limit the type of data that can go into it. They are used to ensure the accuracy and reliability of the data.
Common Constraints:

- PRIMARY KEY: Uniquely identifies each record.
- FOREIGN KEY: Enforces a link between two tables.
- NOT NULL: Ensures a column cannot have a NULL value.
- UNIQUE: Ensures all values in a column are different.
- CHECK: Ensures that all values in a column satisfy a specific condition.
- DEFAULT: Sets a default value for a column if no value is specified.

Question 11: What is normalization?

Theory
Normalization is the process of organizing the columns and tables of a relational database to minimize **data redundancy** and improve **data integrity**. It involves dividing larger tables into smaller, well-structured tables and defining relationships between them.
Normal Forms: The process is guided by a series of "normal forms," such as:

- **1NF (First Normal Form)**: Ensures that table cells hold atomic values and each record is unique.
- **2NF (Second Normal Form)**: Must be in 1NF and all non-key attributes must be fully dependent on the primary key.
- **3NF (Third Normal Form)**: Must be in 2NF and have no transitive dependencies.

Question 12: What is a NULL value?

Theory
A NULL value in SQL represents a **missing or unknown value**. It is not the same as a zero (0) or an empty string (''). A field with a NULL value is a field with no value. Any arithmetic operation performed with NULL results in NULL.

Question 13: What is the difference between CHAR and VARCHAR?

Theory

- **CHAR(n)**: A **fixed-length** string data type. It always reserves n bytes of storage, regardless of the actual length of the string stored. If the string is shorter than n, it is padded with spaces.

    - **Use Case**: Best for data that has a consistent length, like state abbreviations ('CA', 'NY').
-
- **VARCHAR(n)**: A **variable-length** string data type. It only uses the amount of storage needed for the actual string, plus a small overhead to store the length.
    - **Use Case**: Best for data where the length varies, like names or addresses.
-

Question 14: What is the SELECT statement?

Theory
The SELECT statement is the most commonly used DML command. It is used to query the database and retrieve data from one or more tables that matches criteria that you specify.

Question 15: What is the WHERE clause?

Theory
The WHERE clause is used with SELECT, UPDATE, and DELETE statements to **filter records**. It specifies a condition that must be met for a record to be included in the result set or to be modified/deleted.

Question 16: What is the ORDER BY clause?

Theory
The ORDER BY clause is used with the SELECT statement to sort the result set in ascending (ASC, the default) or descending (DESC) order based on one or more columns.

Question 17: What is the DISTINCT keyword?

Theory
The DISTINCT keyword is used with the SELECT statement to return only **unique** values from a column. It eliminates duplicate rows from the result set.

Question 18: What are aliases in SQL?

Theory
Aliases are used to give a table or a column in a table a temporary, alternative name. They are

often used to make column names more readable or to shorten table names in a query, especially in joins. The AS keyword is used to create an alias.

Question 19: What is the BETWEEN operator?

Theory
The BETWEEN operator is used in a WHERE clause to select values within a given range. The range is inclusive: value BETWEEN low AND high.

Question 20: What is the LIKE operator?

Theory
The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. It is used with wildcard characters.

Question 21: What is the IN operator?

Theory
The IN operator is used in a WHERE clause to specify multiple values. It is a shorthand for multiple OR conditions. column IN (value1, value2, ...)

Question 22: What are wildcard characters in SQL?

Theory
Wildcard characters are used with the LIKE operator to substitute for one or more characters in a string search.

- **% (Percent sign)**: Represents zero, one, or multiple characters.
- **_ (Underscore)**: Represents a single character.

Question 23: What is the difference between COUNT(*) and COUNT(column_name)?

Theory

- **COUNT(*)**: Counts the **total number of rows** in a table, regardless of whether they contain NULL values.

- **COUNT(column_name)**: Counts the number of rows where the specified **column_name is NOT NULL**. It ignores NULL values in that column.

Intermediate SQL Questions (25 Questions)

Joins and Advanced Queries

Question 24: What are the different types of joins in SQL?

Theory
The main types of SQL joins are:

1. **INNER JOIN**: Returns records that have matching values in both tables.
2. **LEFT JOIN (or LEFT OUTER JOIN)**: Returns all records from the left table, and the matched records from the right table. The result is NULL from the right side if there is no match.
3. **RIGHT JOIN (or RIGHT OUTER JOIN)**: Returns all records from the right table, and the matched records from the left table. The result is NULL from the left side if there is no match.
4. **FULL OUTER JOIN**: Returns all records when there is a match in either the left or right table. It combines the results of both LEFT and RIGHT joins.
5. **CROSS JOIN**: Returns the Cartesian product of the two tables (all possible combinations of rows).
6. **SELF JOIN**: A regular join, but the table is joined with itself.

Question 25: Explain INNER JOIN with an example.

Theory
An INNER JOIN selects records that have matching values in both tables being joined. It is the most common type of join.

Example
Given a Customers table and an Orders table:

code SQL
downloadcontent_copyexpand_less
```
  SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

This query will return only the orders that belong to an existing customer. If an order had a CustomerID that did not exist in the Customers table, it would not be included in the result.

Question 26: What is LEFT JOIN (LEFT OUTER JOIN)?

Theory
A LEFT JOIN returns **all records from the left table** (the first table mentioned), and the matched records from the right table. If there is no match for a record from the left table in the right table, the columns from the right table will contain NULL.

Use Case: Finding all customers and any orders they might have. This would include customers who have never placed an order.

Question 27: What is RIGHT JOIN (RIGHT OUTER JOIN)?

Theory
A RIGHT JOIN is the opposite of a LEFT JOIN. It returns **all records from the right table** (the second table mentioned), and the matched records from the left table. If there is no match, the columns from the left table will be NULL.

Use Case: Finding all orders and the customers who placed them. This would include any orders that might have a CustomerID which, for some reason, is not in the Customers table.

Question 28: What is FULL OUTER JOIN?

Theory
A FULL OUTER JOIN combines the results of both LEFT JOIN and RIGHT JOIN. It returns all records from both tables. It will place NULLs in the columns of the table where there is no match.

Use Case: To see a complete list of all customers and all orders, and to see which customers have no orders and which orders have no customers.

Question 29: What is a CROSS JOIN?

Theory
A CROSS JOIN produces the **Cartesian product** of two tables. This means it returns a result set where each row from the first table is combined with every row from the second table. It does not use an ON clause.

Use Case: It is rarely used in practice, but can be useful for generating a large amount of test data or for creating a list of all possible pairings (e.g., pairing every customer with every product).

Question 30: What is a SELF JOIN?

Theory
A SELF JOIN is a join in which a table is joined with itself. To do this, you must use aliases to give the table two different temporary names in the query, so that you can treat it as two separate tables.

Use Case: To find relationships between records within the same table, such as finding all employees who have the same manager, or finding customers who live in the same city.

Question 31: What are aggregate functions?

Theory
Aggregate functions perform a calculation on a set of values and return a single, summary value. They are often used with the GROUP BY clause.
Common Aggregate Functions:

- COUNT(): Counts the number of rows.
- SUM(): Calculates the sum of values.
- AVG(): Calculates the average of values.
- MIN(): Returns the minimum value.
- MAX(): Returns the maximum value.

Question 32: Explain the GROUP BY clause.

Theory
The GROUP BY clause is used with aggregate functions to group rows that have the same values in specified columns into summary rows. It groups the data so that the aggregate function can be applied to each group.

Example: To find the number of orders for each customer:

code SQL
downloadcontent_copyexpand_less
IGNORE_WHEN_COPYING_START
IGNORE_WHEN_COPYING_END

```
  SELECT CustomerID, COUNT(OrderID)
FROM Orders
GROUP BY CustomerID;
```

Question 33: What is the HAVING clause?

Theory
The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions. HAVING is used to **filter the results of a GROUP BY** based on the result of an aggregate function.

Question 34: What is the difference between WHERE and HAVING?

Theory

- **WHERE clause**: Filters **rows** *before* any grouping or aggregation is performed. It operates on individual row data.
- **HAVING clause**: Filters **groups** *after* the GROUP BY aggregation has been performed. It operates on the output of aggregate functions.

Order of Operations: FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY.

Question 35: What are subqueries?

Theory
A subquery (also called a nested query or inner query) is a SELECT statement that is nested inside another SQL statement (like SELECT, INSERT, UPDATE, or DELETE, or inside another subquery). The subquery is executed first, and its result is used by the outer query.

Question 36: What is a correlated subquery?

Theory
A correlated subquery is a subquery that depends on the outer query for its values. Unlike a regular subquery, which can be run independently, a correlated subquery is evaluated **once for each row** processed by the outer query. This can make them computationally expensive.

Question 37: What is the difference between correlated and non-correlated subqueries?

Theory

- **Non-correlated (or Nested) Subquery**:
    - **Execution**: The inner query is executed **once**, and its result is used by the outer query.
    - **Dependency**: The inner query is independent and can be run on its own.
-
- **Correlated Subquery**:
    - **Execution**: The inner query is executed **repeatedly**, once for each row of the outer query.
    - **Dependency**: The inner query depends on the current row of the outer query.
-

Question 38: What is UNION and UNION ALL?

Theory
The UNION and UNION ALL operators are used to combine the result sets of two or more SELECT statements. The columns in the SELECT statements must have similar data types.

Question 39: What is the difference between UNION and UNION ALL?

Theory

- **UNION**: Combines the result sets and removes **duplicate** rows.
- **UNION ALL**: Combines the result sets but **includes all duplicate** rows.

Performance: UNION ALL is significantly faster than UNION because it does not have to perform the extra work of checking for and removing duplicates.

Question 40: What are indexes?

Theory
An index is a special lookup table that the database search engine can use to speed up data retrieval. It is a data structure (often a B-tree) that improves the speed of operations on a database table at the cost of additional writes and storage space.

Question 41: What is the difference between clustered and non-clustered indexes?

Theory

- **Clustered Index**:
    - **Physical Order**: A clustered index determines the **physical order** of data in a table. The rows on the disk are stored in the order specified by the clustered index.
    - **Quantity**: A table can have **only one** clustered index.
    - **Analogy**: Like the words in a physical dictionary, which are physically sorted alphabetically.
-
- **Non-clustered Index**:
    - **Logical Order**: A non-clustered index has a structure separate from the data rows. It contains the index key values, and each key has a pointer to the data row that contains that value.
    - **Quantity**: A table can have **multiple** non-clustered indexes.
    - **Analogy**: Like the index at the back of a book. The index is sorted alphabetically, but the pages it points to are not.
-

Question 42: What are views?

Theory
A view is a **virtual table** based on the result-set of a SELECT statement. It contains rows and columns, just like a real table, but it does not store the data itself. The data is generated dynamically when the view is queried.
Uses:

- To simplify complex queries.
- To restrict access to data (e.g., show only certain columns or rows to a user).

Question 43: What is a stored procedure?

Theory
A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. It is a set of SQL statements that can be executed as a single unit.
Uses:

- To encapsulate and share business logic.
- To improve performance (the execution plan can be cached).
- To enhance security by granting users permission to execute the procedure instead of the underlying tables.

Question 44: What is a function in SQL?

Theory
A function in SQL is a block of code that performs a specific task and **must return a single value**. There are two types:

1. **System-defined functions**: Built-in functions like SUM(), GETDATE().
2. **User-defined functions (UDFs)**: Functions created by the user to perform custom logic.

Question 45: What is the difference between a stored procedure and a function?

Theory
Feature	Stored Procedure	User-Defined Function
Return Value	**Not required** to return a value. Can return zero, one, or multiple values via output parameters.	**Must return** a single value.
Invocation	Can be called independently using EXEC or EXECUTE.	Can be called from within a SELECT statement, just like a built-in function.
Operations	Can perform DML operations (INSERT, UPDATE, DELETE).	Is generally restricted to SELECT statements (cannot modify data).
Transactions	Can be used to manage transactions.	Cannot be used to manage transactions.

Question 46: What are triggers?

Theory
A trigger is a special type of stored procedure that is **automatically executed** in response to a specific DML event (INSERT, UPDATE, or DELETE) on a table.
Use Case: To enforce complex business rules or to maintain data integrity that cannot be handled by simple constraints. For example, a trigger could be used to create an audit trail by inserting a record into a log table every time a row in another table is updated.

Question 47: What is a cursor?

Theory
A cursor is a database object that allows you to traverse the rows of a result set one by one. It is a control structure that enables **row-by-row processing**.
Use Case: While set-based operations are almost always preferred in SQL for performance, cursors are sometimes necessary for complex, iterative procedural logic that cannot be

expressed in a single SQL statement. Their use should be minimized as they are generally slower.

Question 48: What is a CTE (Common Table Expression)?

Theory
A CTE is a temporary, named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH clause.
Uses:

- To break down complex queries into simpler, logical, and more readable steps.
- To enable recursive queries, such as traversing a hierarchical data structure (e.g., an organizational chart).

Of course. Here is a comprehensive list of answers to the SQL interview questions, formatted for clarity and interview-readiness.

Performance and Complex Queries

Question 49: What are window functions?

Theory
Window functions are a powerful feature in modern SQL that perform a calculation across a set of table rows that are somehow related to the current row. Unlike regular aggregate functions, which collapse the rows into a single output row, window functions perform a calculation for each row based on a "window" of related rows and do not reduce the number of rows in the output.

Explanation
They are used for tasks like calculating running totals, moving averages, or ranking rows within a group. A window function is defined using the `OVER()` clause.

Question 50: Explain ROW_NUMBER(), RANK(), and DENSE_RANK().

Theory
These are three common ranking window functions that assign a rank to each row within a partition of a result set.

- **`ROW_NUMBER()`:** Assigns a **unique sequential integer** to each row within the partition. No two rows will have the same number.

- **`RANK()`**: Assigns a rank to each row. If there are **`ties`** in the ordering value, the tied rows receive the same rank, and a **`gap is left`** in the sequence for the next rank. (e.g., 1, 2, 2, 4).
- **`DENSE_RANK()`**: Similar to RANK(), but if there are ties, the tied rows receive the same rank, and **`no gap is left`** in the sequence. (e.g., 1, 2, 2, 3).

Question 51: What is PARTITION BY?

Theory
The `PARTITION BY` clause is used within the `OVER()` clause of a window function. It divides the rows of the result set into partitions (groups) to which the window function is applied.

Explanation
It is similar to `GROUP BY`, but it does not collapse the rows. If you use `SUM(Sales) OVER (PARTITION BY Country)`, the function will calculate the total sales for each country, but it will return this same total value on every single row belonging to that country, without aggregating the rows themselves.

Question 52: What are materialized views?

Theory
A materialized view is a database object that contains the results of a query, just like a regular view. However, unlike a regular view where the query is run every time it is accessed, a materialized view **stores the result set as a physical table** on the disk.

Explanation
- **Performance**: They can significantly improve performance for complex and expensive queries, as the data is pre-computed.
- **Data Freshness**: The data in a materialized view is not always up-to-date. It needs to be **refreshed** periodically to reflect the changes in the underlying base tables.

Question 53: What is database partitioning?

Theory
Database partitioning is the process of dividing a very large table into smaller, more manageable pieces, called partitions, while still treating it as a single table logically.
Types:

- **Horizontal Partitioning (Sharding)**: Splits the table by rows. For example, a `Sales` table could be partitioned by year, with each partition containing the sales data for a specific year.
- **Vertical Partitioning**: Splits the table by columns. For example, columns with large text or blob data could be moved to a separate partition.

Benefit: It can dramatically improve query performance and manageability, as the database can often scan only the relevant partitions instead of the entire table.

Question 54: What is a transaction?

Theory
A transaction is a sequence of one or more SQL operations that are executed as a **single, atomic unit of work**. All the operations within a transaction must either succeed completely, or they must all fail, leaving the database in its original state.

Question 55: What are ACID properties?

Theory
ACID is an acronym that describes the four key properties that guarantee the reliability of database transactions.

1. **Atomicity**: Ensures that all operations within a transaction are completed successfully as a single, indivisible unit. If any part of the transaction fails, the entire transaction is rolled back.
2. **Consistency**: Ensures that a transaction brings the database from one valid state to another. All data integrity constraints must be satisfied.
3. **Isolation**: Ensures that concurrent transactions do not interfere with each other. The intermediate state of a transaction is not visible to other transactions until it is committed.
4. **Durability**: Ensures that once a transaction has been successfully committed, it will remain so, even in the event of a system failure (like a power outage or crash).

Question 56: What is a deadlock?

Theory
A deadlock is a situation that occurs in a database when two or more transactions are waiting for each other to release a lock on a resource.
Example:

1. Transaction A locks Resource X and requests a lock on Resource Y.

2. At the same time, Transaction B locks Resource Y and requests a lock on Resource X. Both transactions are now stuck in a circular wait, and neither can proceed.

Question 57: How do you handle deadlocks?

Theory
Relational database management systems have built-in mechanisms to handle deadlocks.

1. **Deadlock Detection**: The RDBMS periodically runs a "waits-for" graph algorithm to detect if a circular chain of transactions exists.
2. **Victim Selection**: If a deadlock is detected, the system will choose one of the transactions as a "victim."
3. **Rollback**: The victim transaction is **aborted and rolled back**, which releases its locks and allows the other transaction(s) to proceed. The application that initiated the victim transaction will receive an error and typically must retry the transaction.

Question 58: What is query optimization?

Theory
Query optimization is the process by which the database management system determines the most efficient way to execute a given SQL query. The component of the RDBMS that does this is called the **query optimizer**.

Explanation
For a single SQL query, there are often many different possible execution plans (e.g., which index to use, what join algorithm to use, the order of joins). The optimizer evaluates these different plans, estimates their cost (in terms of I/O, CPU, etc.), and chooses the one with the lowest estimated cost.

Question 59: How do you optimize slow queries?

Theory
Optimizing slow queries is a critical task for a database administrator or data engineer.
Strategies:

1. **Ensure Proper Indexing**: This is the most common and effective solution. Check if the columns used in `WHERE` clauses, `JOIN` conditions, and `ORDER BY` clauses are properly indexed.

2. **Analyze the Execution Plan**: Use the `EXPLAIN` (or `EXPLAIN PLAN`) command to see how the database is executing the query. Look for full table scans on large tables, as this is often a sign of a missing index.
3. **Rewrite the Query**:
    a. Avoid using `SELECT *`; only select the columns you need.
    b. Be careful with `LIKE` clauses that start with a wildcard (`%text`), as this often prevents the use of an index.
    c. Break down complex queries into simpler steps using temporary tables or CTEs.
4. **Avoid Correlated Subqueries**: If possible, rewrite correlated subqueries as regular joins or CTEs, as they can be very slow.
5. **Keep Statistics Updated**: Ensure that the database has up-to-date statistics about the data distribution, as the optimizer relies on these to make good decisions.

Question 60: What is SQL injection?

Theory
SQL injection is a common web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It is typically caused by an application that insecurely concatenates user input directly into an SQL statement. This can allow an attacker to execute arbitrary SQL code, potentially leading to data theft, modification, or destruction.

Question 61: How do you prevent SQL injection?

Theory
The primary way to prevent SQL injection is to **never trust user input** and to **never concatenate it directly into SQL strings**.
Methods:

1. **Use Prepared Statements (with Parameterized Queries)**: This is the most effective defense. The SQL query is sent to the database first with placeholders (like `?` or `:name`), and then the user-supplied values are sent separately. The database treats the user input as data only, not as executable code, which completely prevents injection.
2. **Use Stored Procedures**: Stored procedures can also help, as the logic is pre-compiled on the database.
3. **Input Validation**: Sanitize and validate all user input to ensure it conforms to the expected format (e.g., checking that a number is actually a number).
4. **Principle of Least Privilege**: Ensure that the database account used by the application has the minimum necessary permissions, so that even if an injection attack is successful, the damage is limited.

Question 62: What is the execution plan?

Theory
An execution plan (or query plan) is a sequence of steps that the database query optimizer generates to execute an SQL statement. It is the roadmap that the database will follow to retrieve the requested data.

Explanation
The plan details operations like:

- Which tables to access and in what order.
- Whether to use an index scan or a full table scan.
- Which join algorithm to use (e.g., nested loop, hash join, merge join).
- How to perform aggregations and sorting.
  Analyzing the execution plan using the `EXPLAIN` command is the key to diagnosing and fixing slow queries.

Question 63: What are temporary tables?

Theory
A temporary table is a table that is created and exists only for the duration of a database session or a specific transaction. It is automatically dropped when the session ends.
Use Case: They are used to store intermediate results in a complex multi-step process. They can be very useful for breaking down a large, complicated query into a series of simpler, more manageable steps, which can sometimes improve performance and readability.

Question 64: What is the difference between DELETE, TRUNCATE, and DROP?

Theory

Command	`DELETE`	`TRUNCATE`	`DROP`
Operation	**`DML.`** `Removes rows from a table one by one.`	**`DDL.`** `Deallocates the data pages, removing all rows quickly.`	**`DDL.`** `Removes the entire table, including its structure.`
WHERE Clause	**`Can use a`** `WHERE` **`clause to remove`**	**`Cannot use a`** `WHERE` **`clause. It`**	`-`

	`specific rows.`	`removes all rows.`	
Logging	`Logs each row deletion. Can be slow for large tables.`	`Minimal logging. Very fast.`	`-`
Triggers	`Can fire DELETE triggers for each row.`	`Does not fire DELETE triggers.`	`-`
Rollback	`Can be rolled back.`	`Cannot be rolled back (in most systems).`	`Cannot be rolled back.`
Identity Reset	`Does not reset the identity column.`	`Resets the identity column.`	`-`

Question 65: What is a database backup?

Theory

A database backup is a copy of the data and schema of a database that can be used to restore the database to a previous state in the event of a failure, corruption, or disaster. It is a critical component of a disaster recovery plan.

Question 66: What are the different types of backups?

Theory

1. **Full Backup**: A complete copy of the entire database.
2. **Differential Backup**: A copy of only the data that has changed since the **last full backup**. To restore, you need the last full backup and the latest differential backup.
3. **Incremental Backup**: A copy of only the data that has changed since the **last backup of any type** (full or incremental). To restore, you need the last full backup and all subsequent incremental backups.
4. **Transaction Log Backup**: A copy of the transaction log, which contains all the transactions that have occurred since the last log backup. This is used for point-in-time recovery.

Question 67: What is database replication?

Theory
Database replication is the process of creating and maintaining multiple copies of a database on different servers. The servers are known as replicas.
Uses:

- **High Availability**: If the primary server fails, a replica can be promoted to take its place, minimizing downtime.
- **Read Scalability**: Read queries can be distributed across multiple replicas to handle a high volume of read traffic.
- **Disaster Recovery**: A replica can be maintained in a remote geographic location.

Question 68: What is sharding?

Theory
Sharding is a type of database partitioning that separates a very large database into smaller, faster, more manageable parts, called shards. Each shard is an independent database, and together they make up the logical whole.
How it Works: Sharding is a form of **horizontal partitioning**. The data is split across multiple servers based on a "shard key." For example, a `Users` table could be sharded by `UserID`, with users 1-1M on Server A, users 1M-2M on Server B, and so on.

Use Case: It is a key technique for achieving massive **horizontal scalability**, used by large-scale applications like Facebook and Twitter.

Question 69: What are recursive queries?

Theory
A recursive query is a query that refers to itself. They are used to query hierarchical or graph-like data structures, such as an organizational chart (employees and their managers) or a bill of materials.
Implementation: Recursive queries are implemented using **Common Table Expressions (CTEs)**. A recursive CTE has two parts:

1. **Anchor Member**: A `SELECT` statement that returns the base result set (e.g., the top-level CEO of a company).
2. **Recursive Member**: A `SELECT` statement that refers to the CTE itself and is joined with the anchor member to find the next level of the hierarchy. This is combined with the anchor using a `UNION ALL`.

Question 70: What is the MERGE statement?

Theory
The `MERGE` statement (sometimes called an `UPSERT`) is a powerful command that performs `INSERT`, `UPDATE`, or `DELETE` operations on a target table based on its join with a source table. It allows you to synchronize two tables in a single statement.
Logic:

- `WHEN MATCHED`: Defines the action to take (e.g., `UPDATE`) when a row from the source matches a row in the target.
- `WHEN NOT MATCHED BY TARGET`: Defines the action to take (e.g., `INSERT`) when a row exists in the source but not the target.
- `WHEN NOT MATCHED BY SOURCE`: Defines the action to take (e.g., `DELETE`) when a row exists in the target but not the source.

Practical SQL Questions (15 Questions)

Real-World Query Problems

Question 73: Write a query to find duplicate records.

Strategy: Use `GROUP BY` on the columns you want to check for duplicates and then use `HAVING` to filter for groups with a count greater than 1.

```
SELECT
    email,
    COUNT(email)
FROM
    Users
GROUP BY
    email
HAVING
    COUNT(email) > 1;
```

Question 74: Write a query to find the second highest salary.

Strategy: Use `DENSE_RANK()` or `RANK()` to rank the salaries and then select the one with a rank of 2. `DENSE_RANK` is often preferred to handle ties without creating gaps.

```
WITH RankedSalaries AS (
```

```
    SELECT
        Salary,
        DENSE_RANK() OVER (ORDER BY Salary DESC) as SalaryRank
    FROM
        Employees
)
SELECT
    Salary
FROM
    RankedSalaries
WHERE
    SalaryRank = 2;
```

Alternative (less robust):

```
SELECT MAX(Salary)
FROM Employees
WHERE Salary < (SELECT MAX(Salary) FROM Employees);
```

Question 75: Write a query to find employees with a salary greater than their manager's.

Strategy: This requires a `SELF JOIN` on the `Employees` table, joining the table to itself on the `ManagerID` column.

```
SELECT
    E.Name AS EmployeeName,
    E.Salary AS EmployeeSalary,
    M.Name AS ManagerName,
    M.Salary AS ManagerSalary
FROM
    Employees E
JOIN
    Employees M ON E.ManagerID = M.EmployeeID
WHERE
    E.Salary > M.Salary;
```

Question 76: Write a query to find the top 5 customers by total order amount.

Strategy: Join `Customers` and `Orders` tables, `GROUP BY` the customer, `SUM` their order amounts, `ORDER BY` the sum in descending order, and take the top 5 using `LIMIT`.

```sql
SELECT
    C.CustomerName,
    SUM(O.OrderAmount) AS TotalSpent
FROM
    Customers C
JOIN
    Orders O ON C.CustomerID = O.CustomerID
GROUP BY
    C.CustomerName
ORDER BY
    TotalSpent DESC
LIMIT 5;
```

Question 77: Write a query to calculate a running total.

Strategy: Use the `SUM()` window function with an `ORDER BY` clause inside the `OVER()` clause.

```sql
SELECT
    OrderDate,
    OrderAmount,
    SUM(OrderAmount) OVER (ORDER BY OrderDate) AS RunningTotal
FROM
    Sales;
```

Question 78: Write a query to find employees who joined in the last 6 months.

Strategy: Use the `WHERE` clause to filter the `JoinDate` column based on the current date. The exact function for the current date varies (`GETDATE()`, `NOW()`, `CURRENT_DATE`).

```sql
-- For SQL Server
SELECT Name, JoinDate
FROM Employees
WHERE JoinDate >= DATEADD(month, -6, GETDATE());

-- For PostgreSQL/MySQL
SELECT Name, JoinDate
FROM Employees
WHERE JoinDate >= CURDATE() - INTERVAL '6' MONTH;
```

Question 79: Write a query to list departments with more than 10 employees.

Strategy: Use `GROUP BY` on the `DepartmentID` and `HAVING` to filter the groups based on the `COUNT` of employees.

```
SELECT
    D.DepartmentName,
    COUNT(E.EmployeeID) AS NumberOfEmployees
FROM
    Departments D
JOIN
    Employees E ON D.DepartmentID = E.DepartmentID
GROUP BY
    D.DepartmentName
HAVING
    COUNT(E.EmployeeID) > 10;
```

Question 80: Write a query to find common records between two tables.

Strategy: The `INTERSECT` operator is designed for this. It returns the distinct rows that are present in both result sets.

```
SELECT CustomerID, CustomerName FROM ActiveCustomers
INTERSECT
SELECT CustomerID, CustomerName FROM NewCustomers;
```

Alternative (using `INNER JOIN`):

```
SELECT DISTINCT
    A.CustomerID, A.CustomerName
FROM
    ActiveCustomers A
INNER JOIN
    NewCustomers N ON A.CustomerID = N.CustomerID;
```

Question 81: Write a query to remove duplicate records.

Strategy: Use a CTE with the `ROW_NUMBER()` window function to identify and then delete the duplicate rows.

```
WITH NumberedRows AS (
    SELECT
        *,
        ROW_NUMBER() OVER (PARTITION BY email ORDER BY UserID) as rn
    FROM
        Users
)
DELETE FROM NumberedRows WHERE rn > 1;
```

This query partitions the data by `email` (the column with duplicates) and assigns a row number to each. Any row with a row number greater than 1 is a duplicate and is deleted.

Question 82: Write a query to calculate the average salary by department.

Strategy: Use the `AVG()` aggregate function with `GROUP BY` on the department.

```
SELECT
    D.DepartmentName,
    AVG(E.Salary) AS AverageSalary
FROM
    Departments D
JOIN
    Employees E ON D.DepartmentID = E.DepartmentID
GROUP BY
    D.DepartmentName;
```

Question 86: Write a query to find the nth highest salary.

Strategy: This is a generalization of the "second highest salary" problem. Use a CTE with `DENSE_RANK()` and select the desired rank `n`.

```
-- For n = 3 (3rd highest salary)
WITH RankedSalaries AS (
    SELECT
        Salary,
```

```
        DENSE_RANK() OVER (ORDER BY Salary DESC) as SalaryRank
    FROM
        Employees
)
SELECT
    Salary
FROM
    RankedSalaries
WHERE
    SalaryRank = 3;
```

Question 87: Write a query to find employees without managers.

Strategy: An employee without a manager will typically have a `NULL` value in their `ManagerID` column.

```
SELECT
    Name
FROM
    Employees
WHERE
    ManagerID IS NULL;
```

Of course. Here is a comprehensive list of answers to the Git interview questions, formatted for clarity and interview-readiness.

Basic Git Questions (20 Questions)

Version Control Fundamentals

Question 88: What is Git?

Theory
Git is a **distributed version control system (DVCS)**. It is a tool that allows developers to track changes in their source code over time. It helps teams collaborate on projects by allowing them to work on the same codebase simultaneously without overwriting each other's work.

Question 89: What is version control?

Theory
Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows you to:

- Revert files back to a previous state.
- Revert the entire project back to a previous state.
- Compare changes over time.
- See who last modified something that might be causing a problem.
- Collaborate easily on a project.

Question 90: What is a repository in Git?

Theory
A Git repository (or "repo") is a directory that contains all the files and folders for a project, along with the entire history of changes made to those files. This history is stored in a subdirectory named `.git`.

Question 91: What is the difference between Git and GitHub?

Theory

- **Git**: is the **tool** itself. It is a distributed version control system that runs on your local machine to track changes and manage your code history.
- **GitHub**: is a **service**. It is a cloud-based hosting platform for Git repositories. It provides a web interface and collaboration features like pull requests, issue tracking, and code reviews on top of the Git functionality. Other similar services include GitLab and Bitbucket.

Analogy: Git is like the engine of a car. GitHub is the car itself, with a steering wheel, seats, and a nice paint job, all built around the engine.

Question 92: What is the difference between Git and SVN?

Theory

Feature	Git (Distributed - DVCS)	SVN (Centralized - CVCS)
Model	**Distributed**. Every developer	**Centralized**. There is one

	has a full copy of the entire repository history on their local machine.	single, central server that holds the repository. Developers "check out" a working copy.
Offline Work	**Can commit, create branches, view history, and perform most operations offline.**	**Requires a network connection to the central server for almost all operations.**
Branching	**Branching is extremely lightweight and is a core part of the workflow.**	**Branching is a more heavyweight operation and is less commonly used.**
Speed	**Generally much faster, as most operations are performed locally.**	**Slower, as it needs to communicate with the central server.**

Question 93: How do you initialize a Git repository?

Theory
To initialize a Git repository in a new or existing directory, you navigate to that directory in your terminal and run the command:

```
git init
```

This creates a new subdirectory named `.git` that contains all the necessary repository files.

Question 94: What is a commit in Git?

Theory
A commit is a **snapshot** of your repository at a specific point in time. It is the fundamental unit of change in Git. Each commit has a unique ID (a SHA-1 hash) and is associated with a commit message that describes the changes made.

Question 95: How do you clone a repository?

Theory

To create a local copy of a remote repository, you use the `git clone` command followed by the URL of the repository.

```
git clone <repository_url>
```

This creates a new directory on your local machine, copies all the files and the entire commit history from the remote repository, and automatically sets up a tracking connection to the remote.

Question 96: What is the staging area?

Theory
The staging area (also known as the "index") is an intermediate area in Git. It is a file that stores information about what will go into your next commit.
Workflow: You first add your file changes from the **working directory** to the **staging area**, and then you commit the changes from the staging area to the **repository**. This allows you to carefully craft your commits, including only the specific changes you want.

Question 97: What is the working directory?

Theory
The working directory (or working tree) is the directory on your file system that contains the actual files of your project. It is the version of the project that you are currently working on and modifying.

Question 98: What is HEAD in Git?

Theory
`HEAD` is a reference or a pointer that points to the **current commit** you are working on. It is a symbolic name for the last commit in the branch you are currently checked out to. When you make a new commit, `HEAD` moves forward to point to that new commit.

Question 99: What are the three states of files in Git?

Theory
A file in your working directory can be in one of three main states:

1. **Modified**: The file has been changed but has not yet been committed to the database.
2. **Staged**: The modified file has been marked in its current version to go into the next commit snapshot (it has been added to the staging area).
3. **Committed**: The data is safely stored in your local database (the `.git` directory).

(There is also an "untracked" state for files that are new and have not yet been added to Git.)

Question 100: What is the `git config` command?

Theory
The `git config` command is used to view and set Git configuration variables. These variables can be set at three levels:

- `--local`: Specific to the current repository (stored in `.git/config`).
- `--global`: Applies to all repositories for the current user (stored in `~/.gitconfig`).
- `--system`: Applies to the entire system for all users.
  The most common use is to set your user name and email:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

Question 101: How do you check the status of files?

Theory
The `git status` command is used to see the state of the working directory and the staging area. It shows which files are modified, which are staged, and which are untracked.

Question 102: How do you add files to the staging area?

Theory
The `git add` command is used to add file changes from the working directory to the staging area.

```
# Add a specific file
git add <file_name>

# Add all changed files in the current directory and subdirectories
```

```
git add .
```

Question 103: How do you commit changes?

Theory
The `git commit` command takes the files from the staging area and saves a snapshot of them to the repository's history.

```
# Opens a text editor to write a commit message
git commit

# Commits with an inline message
git commit -m "Your descriptive commit message"
```

Question 104: How do you view the commit history?

Theory
The `git log` command is used to view the commit history of the repository. It shows a list of commits in reverse chronological order, including the commit hash, author, date, and commit message.

Question 105: What is a `.gitignore` file?

Theory
A `.gitignore` file is a text file that tells Git which files or directories to intentionally ignore. Files listed in `.gitignore` will not be tracked by Git, meaning they won't be added to the staging area or committed.

Use Case: This is used for files that are generated by your build process, log files, temporary files, or files containing sensitive information like API keys.

Question 106: How do you create a `.gitignore` file?

Theory

You create a file named `.gitignore` in the root directory of your repository. Inside this file, you list the patterns of files and directories you want to ignore, with one pattern per line. For example:

```
# Ignore log files
*.log

# Ignore a specific directory
/build/

# Ignore a specific file
credentials.env
```

Question 107: What is the `git diff` command?

Theory
The `git diff` command is used to see the differences between various states in Git.

- `git diff`: Shows the differences between your **working directory** and the **staging area**.
- `git diff --staged`: Shows the differences between the **staging area** and the **last commit**.
- `git diff <commit1> <commit2>`: Shows the differences between two commits.

Intermediate Git Questions (20 Questions)

Branching and Collaboration

Question 108: What is branching in Git?

Theory
A branch in Git is a lightweight, movable pointer to a commit. It represents an independent line of development. Branching allows you to diverge from the main line of development (`main` or `master` branch) and work on a new feature or a bug fix in isolation without affecting the main codebase.

Question 109: How do you create a new branch?

Theory

You create a new branch using the `git branch` command:

```
git branch <branch-name>
```

This creates the branch but does not switch to it. A common shortcut to create a new branch and immediately switch to it is:

```
git checkout -b <new-branch-name>
```

Question 110: How do you switch between branches?

Theory
You switch between branches using the `git checkout` command:

```
git checkout <branch-name>
```

(In modern Git, `git switch <branch-name>` is the preferred command for this.)

Question 111: What is merging in Git?

Theory
Merging is the process of combining the changes from one branch into another. The `git merge` command is used to integrate the history of an independent branch back into your current branch.

Question 112: What is the difference between merge and rebase?

Theory
Both `merge` and `rebase` are used to integrate changes from one branch into another, but they do so in different ways.

- **`git merge`:**
  - **Action:** Creates a new **"merge commit"** that has two parent commits. It ties together the histories of the two branches.
  - **History:** Preserves the original, parallel history of the branches. The commit graph will look like a fork and a join.
  - **Pro:** It is a non-destructive operation.

- `git rebase:`
    - `Action: Rewrites the commit history. It takes all the commits from one branch and replays them on top of another branch.`
    - `History: Creates a linear, clean history. It looks as if the feature was developed sequentially, not in parallel.`
    - `Con: It rewrites history, which can be dangerous if you rebase a branch that has already been pushed and is being used by others.`

Question 113: What is a merge conflict?

Theory
A merge conflict occurs when you try to merge two branches that have competing changes, and Git cannot automatically determine which change to accept. This typically happens when the same line of the same file has been edited differently in both branches.

Question 114: How do you resolve merge conflicts?

Theory
1. Git will stop the merge process and mark the conflicting files.
2. **Open the conflicting file**: Inside the file, Git will have added conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) that show the changes from both branches.
3. **Edit the file**: You must manually edit the file to resolve the conflict, choosing which version to keep or combining them. You must also remove the conflict markers.
4. **Stage the resolved file**: Use `git add <resolved-file-name>` to mark the conflict as resolved.
5. **Commit the merge**: Run `git commit` to complete the merge.

Question 115: What is `git pull`?

Theory
The `git pull` command is used to update your local repository with changes from a remote repository. It is a compound command that is equivalent to running:
1. `git fetch` (downloads the changes from the remote)
2. `git merge` (merges the downloaded changes into your current local branch)

Question 116: What is `git fetch`?

Theory
The `git fetch` command downloads commits, files, and refs from a remote repository into your local repo. It **only downloads** the new data; it does **not** integrate any of this new data into your local working files.

Question 117: What is the difference between `git pull` and `git fetch`?

Theory

- **`git fetch`**: is a safe, "read-only" operation. It downloads the remote changes but does not modify your local working directory. It allows you to review the changes before deciding to merge them.
- **`git pull`**: is more aggressive. It downloads the remote changes and immediately tries to **merge** them into your current branch. This can lead to unexpected merge conflicts in your working directory.
  Best Practice: Many experienced developers prefer to use git fetch followed by a manual git merge or git rebase to have more control.

Question 118: What is git push?

Theory
The `git push` command is used to upload your local repository content (your committed changes) to a remote repository.

```
git push <remote-name> <branch-name>
```

Question 119: What is a remote repository?

Theory
A remote repository is a version of your project that is hosted on the internet or a network somewhere (e.g., on GitHub). It allows you to collaborate with other developers.

Question 120: How do you add a remote repository?

Theory
You use the `git remote add` command to add a new remote.

```
git remote add <shortname> <url>
# Example:
git remote add origin https://github.com/user/repo.git
```

The shortname `origin` is the conventional name for the primary remote repository.

Question 121: What is forking?

Theory
A fork is a **personal copy** of someone else's repository that lives on your own account on a hosting service like GitHub. Forking allows you to freely experiment with changes without affecting the original project.

Question 122: What is a pull request?

Theory
A pull request (PR) is a mechanism used in collaborative workflows (like on GitHub) to propose changes to a repository.
Workflow:

1. You fork a repository.
2. You create a new branch and make your changes.
3. You push the branch to your fork.
4. You then open a pull request, which asks the original repository's maintainers to **pull** your changes into their project. This opens a discussion and code review forum for the proposed changes.

Question 123: What is cherry-picking?

Theory
Cherry-picking in Git is the act of choosing a single, specific commit from one branch and applying it onto another branch. This is in contrast to a merge or rebase, which apply many commits. The `git cherry-pick <commit-hash>` command is used for this.

Use Case: It is useful for applying a critical bug fix from a development branch to a stable production branch without bringing over all the other new features.

Question 124: What is `git stash`?

Theory
The `git stash` command temporarily shelves (stashes) the changes you have made in your working directory so you can switch to a different task.
Workflow:

1. You are working on a feature but need to switch branches to fix an urgent bug. Your work is not ready to be committed.
2. You run `git stash` to save your changes and clean your working directory.
3. You switch branches, fix the bug, and commit.
4. You switch back to your feature branch and run `git stash pop` to re-apply your stashed changes and continue your work.

Question 125: How do you unstage files?

Theory
To remove a file from the staging area (to "un-add" it), you use the `git reset` command.

```
# Unstage a specific file
git reset HEAD <file_name>

# Unstage all files
git reset
```

This moves the file from the staging area back to the modified state in the working directory.

Question 126: How do you undo the last commit?

Theory
There are several ways, depending on what you want to do.

- **If you want to amend the last commit** (e.g., add a forgotten file or change the message):

```
•
```

```
- git add <forgotten-file>
- git commit --amend
```

- **If you want to undo the commit but keep the changes** in your working directory:

```
- git reset --soft HEAD~1
```

- **If you want to completely discard the commit and all its changes** (destructive):

```
- git reset --hard HEAD~1
```

Question 127: What is `git reset`?

Theory
`git reset` is a powerful command that is used to undo changes. It has three main modes:

- `--soft`: Moves `HEAD` to a previous commit but leaves your staging area and working directory unchanged.
- `--mixed` (Default): Moves `HEAD` and resets the staging area to match the specified commit, but leaves the working directory unchanged.
- `--hard`: Moves `HEAD`, resets the staging area, and **resets the working directory**. This is a destructive command that will discard all uncommitted changes.

Advanced Git Questions (20 Questions)

Complex Git Operations

Question 128: What is `git rebase`?

Theory
`git rebase` is a command used to integrate changes from one branch onto another by **rewriting the commit history**. It works by taking the commits from a feature branch and "replaying" them, one by one, on top of the tip of another branch (like `main`). This results in a clean, linear commit history.

Question 129: When should you use rebase vs merge?

Theory

- **Use `merge` when**:
  - You are merging into a public, shared branch (like `main`).
  - You want to preserve the exact, non-destructive history of the feature branch.
- **Use `rebase` when**:
  - You are working on a **private, local feature branch** and want to clean up your history before merging it.
  - You want to maintain a linear project history.
- **Golden Rule of Rebasing**: **Never rebase a branch that has been pushed to a remote and is being used by other collaborators.** This will rewrite the public history and cause significant problems for your team.

Question 130: What is interactive rebase?

Theory

Interactive rebase (`git rebase -i`) is a powerful tool that allows you to modify your commits before you push them. It opens an editor with a list of the commits you are about to rebase and allows you to perform actions like:

- `reword`: Change the commit message.
- `edit`: Stop to amend the commit.
- `squash`: Combine a commit with the previous one.
- `fixup`: Like squash, but discards the commit's message.
- `reorder`: Change the order of the commits.
- `drop`: Delete a commit.
  It is an essential tool for cleaning up a messy local commit history before creating a pull request.

Question 131: What is a detached HEAD?

Theory

A "detached HEAD" state occurs when you check out a specific commit, a tag, or a remote branch directly, instead of a local branch. In this state, `HEAD` is no longer pointing to the tip of a branch but is pointing directly to a commit.

Implication: If you make new commits in this state, they will not belong to any branch. Once you switch away, these commits can be "lost" and may be garbage collected by Git.

Question 132: How do you fix a detached HEAD?

Theory
If you have made commits in a detached HEAD state that you want to keep, you simply need to create a new branch to point to them.

```
# Create a new branch at your current commit
git branch <new-branch-name>

# Then switch to it
git checkout <new-branch-name>
```

Or, as a shortcut:

```
git checkout -b <new-branch-name>
```

Question 133: What is `git reflog`?

Theory
The `git reflog` (reference log) is a safety net. It records almost every change you make to the `HEAD` in your local repository. It keeps track of when you switch branches, make commits, perform resets, or cherry-pick.

Use Case: It is an essential tool for recovering "lost" commits. If you accidentally perform a hard reset and think you have lost your work, you can use `git reflog` to find the hash of the commit before the reset and restore it.

Question 134: What are Git hooks?

Theory
Git hooks are custom scripts that are automatically run by Git at certain points in its execution, such as before a commit (`pre-commit`), after a commit (`post-commit`), or before a push (`pre-push`).

Use Case: They are used to automate and enforce team policies, such as running a linter or a test suite before allowing a commit, or checking that a commit message follows a specific format.

Question 135: What is `git bisect`?

Theory
`git bisect` is a powerful debugging tool used to find the specific commit that introduced a bug in the project's history.
How it Works: It performs an automated **binary search** on your commit history.

1. You start the process with `git bisect start`.
2. You tell it a "bad" commit (where the bug exists, e.g., `HEAD`) and a "good" commit (a past commit where the bug did not exist).
3. Git then checks out a commit in the middle of this range and asks you to test if it's "good" or "bad."
4. You run your tests and tell Git the result (`git bisect good` or `git bisect bad`).
5. Git then repeats the process on the remaining half of the history until it pinpoints the exact commit where the bug was introduced.

Question 136: How do you revert a commit?

Theory
The `git revert <commit-hash>` command is used to safely undo a commit. It does not delete the original commit from the history. Instead, it creates a **new commit** that introduces the inverse of the changes from the specified commit.

Use Case: This is the safe, non-destructive way to undo a change on a **public, shared branch**, as it does not rewrite the project history.

Question 137: What is the difference between `git reset` and `git revert`?

Theory

- **`git reset`: Rewrites history.** It moves the branch pointer backwards to a previous commit, effectively discarding the commits that came after it. It is a "destructive" operation and should only be used on local, private branches.

- **`git revert`: Creates new history.** It creates a new commit that undoes the changes of a previous commit. It is a "non-destructive" operation and is the safe way to undo changes on a public, shared branch.

Question 147: What are Git workflows?

Theory
A Git workflow is a prescribed recipe or strategy for how a team should use Git to collaborate effectively. It defines a set of rules and conventions, such as how to use branches, when to merge, and how to handle releases.
Common Workflows:

1. **Centralized Workflow**: A simple workflow where everyone works on a single `main` branch.
2. **Feature Branch Workflow**: The most common workflow. All new development is done on dedicated feature branches, which are then merged into `main` via a pull request.
3. **Gitflow Workflow**: A more complex and stricter workflow that uses dedicated branches for features, releases, and hotfixes. It is well-suited for projects with a scheduled release cycle.
4. **Forking Workflow**: Common in open-source projects. Contributors fork the main repository, make changes in their fork, and then submit a pull request back to the original project.

Of course. Here is a comprehensive list of answers to the Git, GitHub, and Programming Basics interview questions, formatted for clarity and interview-readiness.

GitHub-Specific Questions (20 Questions)

GitHub Platform Features

Question 148: What is GitHub?

Theory
GitHub is a **web-based hosting service** for version control using Git. It is a for-profit company that offers a cloud-based Git repository hosting service. Essentially, it provides a platform and a user-friendly web interface for storing, managing, and collaborating on Git repositories.

Key Features:

- Git repository hosting.
- Collaboration features like **Pull Requests**, code reviews, and issue tracking.
- Automation tools like **GitHub Actions** for CI/CD.

- Project management tools like **GitHub Projects**.
- Hosting for documentation and static websites via **GitHub Pages**.

Question 149: What are GitHub Actions?

Theory
GitHub Actions is a powerful and flexible **automation and CI/CD (Continuous Integration/Continuous Delivery)** platform built directly into GitHub. It allows you to automate your software development workflows, such as building, testing, and deploying your code, directly from your repository.

Question 150: What is a workflow in GitHub Actions?

Theory
A **workflow** is a configurable automated process that you define in your repository. It is defined by a **YAML file** located in the `.github/workflows` directory. A workflow is made up of one or more **jobs** and is triggered by specific **events** (like a `push` or a `pull_request`).

Question 151: What are runners in GitHub Actions?

Theory
A **runner** is a server that executes the jobs in your GitHub Actions workflows. It is the machine that runs your automated tasks.
Types:

1. **GitHub-hosted runners**: Virtual machines hosted by GitHub with a pre-installed environment (e.g., Ubuntu, Windows, macOS).
2. **Self-hosted runners**: Your own machines that you can register with GitHub Actions, giving you more control over the hardware and software environment.

Question 152: What are artifacts in GitHub Actions?

Theory
Artifacts are files or a collection of files that are produced during a workflow run. They allow you to persist data after a job has completed and to share data between jobs in the same workflow.

Use Case: Storing build outputs (like a compiled binary or a packaged application) or test reports.

Question 153: How do you manage secrets in GitHub Actions?

Theory
Secrets are encrypted environment variables that you create in a repository or an organization. They are used to store sensitive information, such as API keys, access tokens, or credentials, that you need to use in your workflows.
How it Works:

- Secrets are stored securely in GitHub.
- They are injected into the runner's environment at runtime.
- Their values are automatically redacted in the logs to prevent accidental exposure.

Question 154: What is GitHub Pages?

Theory
GitHub Pages is a static site hosting service that takes HTML, CSS, and JavaScript files directly from a repository on GitHub, optionally runs the files through a build process, and publishes a website. It is commonly used for hosting project documentation, blogs, or personal portfolio websites.

Question 155: What are GitHub Issues?

Theory
GitHub Issues are a built-in feature for tracking tasks, enhancements, and bugs for your projects on GitHub. They are an integrated project management and communication tool.

Question 156: What is GitHub Projects?

Theory
GitHub Projects is a tool that provides a flexible, Kanban-style board to help you organize and prioritize your work. You can link Issues and Pull Requests directly to the project board to track their status.

Question 157: What are GitHub Releases?

Theory
GitHub Releases are a way to package and distribute software to your users. They are based on Git tags and allow you to attach binary files (like compiled executables), release notes, and other assets to a specific version of your code.

Question 158: What is a GitHub Wiki?

Theory
A GitHub Wiki is a section in a repository that can be used to host detailed documentation for your project, such as how-to guides, design documentation, and usage examples. It is a simple way to create and share in-depth content about your project.

Question 159: What are GitHub templates?

Theory
GitHub provides templates for common community health files to streamline their creation. This includes templates for:

- **Issue templates**: Pre-fillable templates for users to report bugs or request features.
- **Pull request templates**: A checklist or template for contributors to fill out when submitting a PR.
- Files like `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`, etc.

There are also **repository templates**, which allow you to turn a repository into a template so that others can generate new repositories with the same directory structure and files.

Question 160: What is branch protection in GitHub?

Theory
Branch protection rules are a feature in GitHub that allows repository administrators to enforce certain workflows for one or more branches.
Common Rules:

- **Require pull request reviews before merging**: Enforce that at least one other person must approve a PR.

- **Require status checks to pass before merging**: Block merging if CI tests (like from GitHub Actions) have not passed.
- **Prevent force pushes**: Disallow rewriting the history of the protected branch.

This is a critical feature for maintaining the stability and quality of important branches like `main`.

Question 161: What are GitHub Apps?

Theory
GitHub Apps are a way to extend and automate your workflow on GitHub. They are first-class actors within GitHub that can be installed directly on organizations and repositories and granted access to specific permissions.
Use Case: They are used to build tools for continuous integration, code analysis, and project management that integrate deeply with the GitHub platform.

Question 162: What is the GitHub API?

Theory
The GitHub API is an interface that allows developers to interact with GitHub programmatically. You can use it to create tools that automate tasks, integrate with other services, and pull data from GitHub. There are two versions: a REST API and a GraphQL API.

Question 163: What are reusable workflows?

Theory
Reusable workflows are a feature in GitHub Actions that allows you to call one workflow from within another workflow. This allows you to avoid duplicating workflow code and to create a centralized, maintainable library of common workflow logic (e.g., a standard `build-and-test` workflow that can be called by multiple other workflows).

Question 164: What is the difference between GitHub-hosted and self-hosted runners?

Theory
- **GitHub-hosted runners**:
    - **Managed by GitHub**.
    - Provide a clean, fresh virtual machine for every job.

    - Come with a pre-installed set of common software and tools.
    - **Pro**: Easy to use, no maintenance.
    - **Con**: Less flexible, can be more expensive for large-scale usage.
- **Self-hosted runners**:
    - **Managed by you**. You install the runner application on your own servers (cloud or on-premise).
    - Give you full control over the hardware, operating system, and software.
    - **Pro**: Highly customizable, more cost-effective for frequent jobs, can access resources in a private network.
    - **Con**: You are responsible for maintenance, security, and updates.

Question 165: How do you define workflow triggers?

Theory
Workflow triggers are defined in the workflow's YAML file using the `on` keyword. They specify the **events** that will cause the workflow to run.
Examples:

```yaml
on: push  # Trigger on any push

on:
  push:
    branches:
      - main  # Trigger only on pushes to the main branch
  pull_request:
    branches:
      - main  # Trigger on pull requests targeting main

on:
  schedule:
    - cron: '0 8 * * 1' # Trigger every Monday at 8 AM UTC
```

Question 166: What are jobs and steps in GitHub Actions?

Theory

- **Job**: A workflow is made up of one or more jobs. By default, jobs run in **parallel**. A job is a set of **steps** that execute on the same runner.
- **Step**: A step is an individual task that can run commands or an **action** (a reusable piece of code). Steps within a job run in **sequence**.

Question 167: How do you use GitHub for CI/CD?

Theory
GitHub Actions is the primary tool for implementing CI/CD on GitHub.

- **Continuous Integration (CI)**:
    - **Goal**: To automatically build and test your code every time a developer pushes a change.
    - **Workflow**: You would create a workflow that is triggered `on: push` or `on: pull_request`. This workflow would have jobs that:
        - Check out the code.
        - Set up the required environment (e.g., install Python).
        - Install dependencies.
        - Run a linter.
        - Run unit tests.
- **Continuous Delivery/Deployment (CD)**:
    - **Goal**: To automatically deploy the application after it has passed the CI tests.
    - **Workflow**: You can add a new job to your workflow that is triggered only after the CI job succeeds. This deployment job would:
        - Build the application (e.g., create a Docker image).
        - Push the artifact to a registry.
        - Deploy the new version to a staging or production environment (using secrets to authenticate with your cloud provider).

Practical Git Scenarios (15 Questions)

Question 168: How do you undo the last commit but keep the changes?

Command

```
git reset --soft HEAD~1
```

Explanation: `--soft` moves the `HEAD` pointer back one commit but leaves the changes from that commit in your working directory and staging area.

Question 169: How do you completely remove a file from Git history?

Command: Use `git filter-branch` or the more modern and recommended `git-filter-repo` tool. This is a destructive operation that rewrites history.

```
# Example using filter-branch (use with caution)
git filter-branch --tree-filter 'rm -f path/to/your/file' HEAD
```

Question 170: How do you rename a branch?

Command:

```
# To rename the current branch
git branch -m <new-name>

# To rename a different branch
git branch -m <old-name> <new-name>
```

Question 171: How do you delete a branch locally and remotely?

Commands:

```
# Delete a local branch
git branch -d <local-branch-name>

# Delete a remote branch
git push origin --delete <remote-branch-name>
```

Question 172: How do you squash commits?

Command: Use **interactive rebase**.

1. `git rebase -i HEAD~N` (where `N` is the number of commits to squash).
2. An editor will open. For the commits you want to squash, change the word `pick` to `squash` (or `s`).
3. Save and close the editor. Another editor will open to let you write the new commit message for the combined commit.

Question 173: How do you move commits from one branch to another?

Command: Use `git cherry-pick`.

1. Check out the branch you want to move the commits *to*.
2. Run `git cherry-pick <commit-hash>` for each commit you want to apply.

Question 174: How do you recover deleted files?

Command:

- **If the file was just deleted but not committed**:

```
git checkout -- <file-name>
```

- **If the deletion was committed**: Find the commit before the deletion using `git log`, and then check out the file from that commit:

```
git checkout <commit-hash-before-delete> -- <file-name>
```

Question 175: How do you find which commit introduced a bug?

Command: Use `git bisect`. It performs an automated binary search through your commit history.

Question 176: How do you change the commit message of the last commit?

Command:

```
git commit --amend
```

This will open your text editor to let you change the message.

Question 177: How do you split a commit into multiple commits?

Command: Use interactive rebase.

1. `git rebase -i HEAD~1` (or the commit you want to split).
2. Change `pick` to `edit` for that commit and save.
3. Git will stop at that commit. Now, use `git reset HEAD~1` to un-commit the changes while keeping them in your working directory.
4. You can now use `git add` and `git commit` to create multiple, smaller commits.
5. Once you are done, run `git rebase --continue`.

Question 178: How do you handle large files in Git?

Tool: Use **Git Large File Storage (LFS)**. Git itself is not designed to handle large binary files. Git LFS is an extension that stores large files on a separate server and keeps lightweight pointers in the Git repository itself.

Programming Basics Questions (121 Questions)

Fundamental Concepts (22 Questions)

Question 183: What is computer programming?

Theory
Computer programming is the process of writing instructions that a computer can execute to perform a specific task. It involves designing and building an executable computer program to accomplish a particular computational result.

Question 184: What is a programming language?

Theory
A programming language is a formal language comprising a set of instructions that produce various kinds of output. It is the vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. Examples include Python, Java, C++, and JavaScript.

Question 185: What is the difference between high-level and low-level languages?

Theory

- **High-Level Language**:
  - **Abstraction**: High level of abstraction from the computer's hardware.
  - **Readability**: Easy for humans to read and write (e.g., Python, Java).
  - **Portability**: Generally portable across different machine architectures.
- **Low-Level Language**:
  - **Abstraction**: Low level of abstraction; closer to the hardware.
  - **Readability**: Harder for humans to read (e.g., Assembly language, Machine code).
  - **Portability**: Not portable; specific to a particular hardware architecture.

Question 186: What is an algorithm?

Theory

An algorithm is a finite sequence of well-defined, step-by-step instructions or rules designed to perform a specific task or solve a particular problem.

Question 187: What are the characteristics of a good algorithm?

Theory

1. **Correctness**: It should produce the correct output for all valid inputs.
2. **Finiteness**: It must terminate after a finite number of steps.
3. **Well-defined**: Each step must be clear and unambiguous.
4. **Efficiency**: It should solve the problem using a reasonable amount of time (time complexity) and memory (space complexity).
5. **Readability**: It should be easy to understand and maintain.

Question 188: What is a flowchart?

Theory

A flowchart is a graphical representation of an algorithm or a process. It uses standard symbols (like ovals, rectangles, and diamonds) connected by arrows to show the sequence of operations and decisions.

Question 189: What is pseudocode?

Theory
Pseudocode is an informal, high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language but is intended for human reading rather than machine reading. It is a way to plan out an algorithm without worrying about the specific syntax of a programming language.

Question 190: What is a compiler?

Theory
A compiler is a special program that translates the entire source code of a program, written in a high-level language, into machine code all at once, before the program is run. The result is an executable file.
Examples of compiled languages: C, C++, Go.

Question 191: What is an interpreter?

Theory
An interpreter is a program that directly executes instructions written in a programming language, without previously compiling them into a machine language program. It translates and executes the source code line by line.
Examples of interpreted languages: Python, JavaScript, Ruby.

Question 192: What is the difference between a compiler and an interpreter?

Theory

Feature	Compiler	Interpreter
Translation	Translates the entire program at once before execution.	Translates and executes the program line by line.
Output	Creates an intermediate object code or executable file.	Does not create an intermediate file.

Speed	The compiled program runs faster.	Slower, as translation happens at runtime.
Error Checking	Checks for all syntax errors at once during compilation.	Reports errors one by one as they are encountered.
Portability	The executable is platform-specific.	The source code is portable, but requires the interpreter to be installed on the target machine.

Question 193: What is machine code?

Theory
Machine code (or machine language) is a set of instructions executed directly by a computer's central processing unit (CPU). It is a sequence of binary digits (0s and 1s) and is the lowest-level programming language.

Question 194: What is assembly language?

Theory
Assembly language is a low-level programming language that has a strong correspondence between the language's statements and the architecture's machine code instructions. It uses mnemonic codes (like `MOV`, `ADD`) to represent machine code instructions, making it more human-readable than raw binary.

Question 195: What is source code?

Theory
Source code is the set of instructions and statements written by a programmer using a human-readable programming language, like Python or Java. It is the code before it is compiled or interpreted.

Question 196: What is object code?

Theory

Object code is the output of a compiler after it has processed the source code. It is an intermediate representation of the code, usually in machine language, that is not yet linked into a final executable file.

Question 197: What is debugging?

Theory
Debugging is the process of finding and resolving defects or problems (bugs) within a computer program that prevent it from operating correctly.

Question 198: What is testing?

Theory
Testing is the process of evaluating a software application to verify that it meets the specified requirements and to identify any defects. It is a process of validation and verification.

Question 199: What is the difference between a syntax error and a logic error?

Theory

- **Syntax Error**: An error in the source code of a program that violates the rules of the programming language. These errors are caught by the compiler or interpreter before the program is run (e.g., a misspelled keyword, a missing parenthesis).
- **Logic Error**: An error in the program's algorithm that causes it to produce incorrect or unexpected results. The program will run without crashing, but it will not do what it was intended to do. These errors are often much harder to find and fix.

Question 200: What is a runtime error?

Theory
A runtime error is an error that occurs during the execution of a program. The program compiles/interprets successfully, but it encounters an issue that it cannot handle while it is running.
Examples: Division by zero, trying to access a file that doesn't exist, running out of memory.

Question 201: What is an IDE?

Theory
An IDE (Integrated Development Environment) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger. Examples include Visual Studio Code, PyCharm, and Eclipse.

Question 202: What is a framework?

Theory
A software framework is a platform for developing software applications. It provides a foundation on which software developers can build programs for a specific platform. It follows the **Inversion of Control (IoC)** principle; the framework calls your code, not the other way around.
Examples: Django, Ruby on Rails, Angular.

Question 203: What is a library?

Theory
A library is a collection of pre-written code (functions, classes, etc.) that can be used by a program. The programmer is in control and calls the library's code when they need it.
Examples: NumPy, Pandas, React.

Question 204: What is an API?

Theory
An API (Application Programming Interface) is a set of rules and definitions that allows different software applications to communicate with each other. It defines the methods and data formats that applications can use to request and exchange information.

Of course. Here is a comprehensive list of answers to the Programming Basics interview questions, formatted for clarity and interview-readiness.

Data Structures (20 Questions)

Data Organization Concepts

Question 205: What is a data structure?

Theory
A data structure is a specialized format for **organizing, processing, retrieving, and storing data**. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. Different kinds of data structures are suited to different kinds of applications.

Question 206: What is an array?

Theory
An array is a **linear data structure** that stores a collection of elements of the **same data type** in a **contiguous block of memory**. Each element is identified by at least one index or key.

Question 207: What is a linked list?

Theory
A linked list is a **linear data structure** where elements are not stored in contiguous memory locations. Instead, each element (called a **node**) contains the data and a **pointer** (or reference) to the next node in the sequence.

Question 208: What is the difference between an array and a linked list?

Theory

Feature	Array	Linked List
Memory Allocation	**Contiguous** (elements are stored next to each other in memory).	**Non-contiguous** (elements can be scattered in memory).
Size	**Fixed size**. The size must be specified at the time of creation.	**Dynamic size**. It can grow or shrink at runtime.
Access Time	**Fast random access**. `O(1)` time to access any element using its index.	**Slow sequential access**. `O(n)` time to access an element, as you must traverse the list from the beginning.

Insertion/Deletion	**Slow**. `O(n)` time, as it may require shifting other elements.	**Fast**. `O(1)` time if you are at the correct node, as it only requires updating pointers.
Storage	**Less memory overhead.**	**More memory overhead due to storing a pointer for each node.**

Question 209: What is a stack?

Theory
A stack is a **linear data structure** that follows a particular order in which operations are performed. The order is **LIFO (Last-In, First-Out)**. The two main operations are `push` (add an element to the top) and `pop` (remove an element from the top).

Question 210: What is LIFO?

Theory
LIFO stands for **Last-In, First-Out**. It is the principle that governs a stack data structure. It means that the last element added to the stack will be the first one to be removed.
Analogy: A stack of plates. You put a new plate on top and you take a plate off the top.

Question 211: What is a queue?

Theory
A queue is a **linear data structure** that follows a **FIFO (First-In, First-Out)** order. The main operations are `enqueue` (add an element to the rear) and `dequeue` (remove an element from the front).

Question 212: What is FIFO?

Theory
FIFO stands for **First-In, First-Out**. It is the principle that governs a queue data structure. It means that the first element added to the queue will be the first one to be removed.
Analogy: A checkout line at a store. The first person in line is the first person to be served.

Question 213: What is a tree?

Theory
A tree is a **non-linear, hierarchical data structure** that consists of nodes connected by edges. Each tree has a **root** node, and every other node has a parent node. Nodes can have zero or more child nodes.

Question 214: What is a binary tree?

Theory
A binary tree is a type of tree data structure in which each node has at most **two children**, which are referred to as the left child and the right child.

Question 215: What is a binary search tree?

Theory
A Binary Search Tree (BST) is a special type of binary tree that satisfies the **binary search property**:

1. The value of all nodes in the **left subtree** of a node is **less than** the value of the node itself.
2. The value of all nodes in the **right subtree** of a node is **greater than** the value of the node itself.
3. Both the left and right subtrees must also be binary search trees.
   This property allows for very fast searching, insertion, and deletion operations (on average `O(log n)`).

Question 216: What is a graph?

Theory
A graph is a **non-linear data structure** consisting of a set of **vertices (or nodes)** and a set of **edges** that connect pairs of vertices. Graphs are used to model relationships and networks.

Question 217: What is a hash table?

Theory
A hash table is a data structure that implements an associative array abstract data type, a structure that can map **keys** to **values**. It uses a **hash function** to compute an index (a "hash") into an array of buckets or slots, from which the desired value can be found. This allows for extremely fast average-case time for lookup, insertion, and deletion (`O(1)`).

Question 218: What is the difference between linear and non-linear data structures?

Theory

- **Linear Data Structures**: The elements are arranged in a **sequential** or linear order. Each element is connected to its previous and next element.
    - **Examples**: Array, Linked List, Stack, Queue.
- **Non-linear Data Structures**: The elements are not arranged in a sequential order. An element can be connected to multiple other elements, representing a hierarchical or network relationship.
    - **Examples**: Tree, Graph.

Question 224: What is recursion?

Theory
Recursion is a programming technique where a function **calls itself** in order to solve a problem. The function must have a **base case** (a condition under which it stops calling itself) to prevent an infinite loop.

Programming Constructs (22 Questions)

Language Building Blocks

Question 225: What is a variable?

Theory
A variable is a symbolic name or identifier that is associated with a value stored in memory. It is a way to label and store data for later use by the program.

Question 226: What is a constant?

Theory
A constant is a variable whose value cannot be changed after it has been assigned.

Question 227: What are data types?

Theory
A data type is a classification that specifies which type of value a variable can hold and what type of mathematical, relational, or logical operations can be applied to it without causing an error.
Examples: `integer`, `float`, `string`, `boolean`.

Question 228: What is type casting?

Theory
Type casting (or type conversion) is the process of converting a variable from one data type to another.

- **Implicit Casting**: Done automatically by the compiler/interpreter.
- **Explicit Casting**: Done manually by the programmer.

Question 229: What is an operator?

Theory
An operator is a symbol that tells the compiler or interpreter to perform a specific mathematical, relational, or logical operation and produce a final result.
Examples: `+`, `-`, `*`, `/`, `==`, `&&`.

Question 233: What is a conditional statement?

Theory
A conditional statement is a programming construct that performs different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

Question 234: What is an `if` statement?

Theory
An `if` statement is the most basic conditional statement. It executes a block of code only if a specified condition is true. It can be extended with `else if` and `else` clauses to handle other conditions.

Question 236: What is a loop?

Theory
A loop is a control flow statement that allows a block of code to be executed repeatedly as long as a certain condition is met.

Question 237: What is a `for` loop?

Theory
A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string) or other iterable objects.

Question 238: What is a `while` loop?

Theory
A `while` loop repeatedly executes a target statement as long as a given condition is true. It checks the condition *before* executing the loop body.

Question 240: What is the difference between a `while` and a `do-while` loop?

Theory

- **`while` loop (Pre-test loop)**: The condition is checked **before** the loop body is executed. If the condition is false initially, the loop body will never be executed.
- **`do-while` loop (Post-test loop)**: The condition is checked **after** the loop body is executed. This means the loop body is guaranteed to be executed at least once. (Note: Python does not have a `do-while` loop).

Question 241: What is a function?

Theory
A function is a named, reusable block of code that is designed to perform a specific task. It can take inputs (parameters) and can return an output.

Question 242: What is a method?

Theory
A method is a function that is **associated with an object**. It is defined inside a class and is called on an instance of that class.

Question 243: What is the difference between a function and a method?

Theory
The key difference is that a method is implicitly passed the object on which it was called (e.g., the `self` argument in Python). A function is not associated with any object. Essentially, a method is a function that belongs to a class.

Question 244: What are parameters and arguments?

Theory

- **Parameter**: The variable listed inside the parentheses in a function's **definition**. It is a placeholder.
- **Argument**: The actual **value** that is sent to the function when it is **called**.

Question 245: What is function overloading?

Theory
Function overloading is a feature where two or more functions can have the same name but different parameters (different number or types of arguments). The compiler or interpreter decides which one to call based on the arguments provided. (Note: Python does not support this in the traditional sense).

Advanced Concepts (22 Questions)

Programming Paradigms

Question 247: What is object-oriented programming?

Theory
Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code. It structures a program by bundling related properties and behaviors into individual, self-contained objects.

Question 248: What is procedural programming?

Theory
Procedural programming is a programming paradigm based on the concept of the "procedure call." It structures a program as a sequence of procedures or functions to be executed. The focus is on the steps and actions.

Question 249: What is functional programming?

Theory
Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It emphasizes the use of **pure functions**, which have no side effects.

Question 256: What is a pointer?

Theory
A pointer is a variable that stores the **memory address** of another variable. It "points" to the location where the data is stored, rather than storing the data itself. (Note: Python does not have pointers in the same way as C/C++, but all variable names in Python are references to objects in memory, which is a similar concept).

Question 257: What is memory management?

Theory

Memory management is the process of controlling and coordinating computer memory, assigning portions called blocks to various running programs to optimize overall system performance. It involves allocating memory to objects and data, and deallocating (freeing) it when it is no longer needed.

Question 258: What is garbage collection?

Theory
Garbage collection (GC) is a form of **automatic memory management**. The garbage collector is a background process that automatically identifies memory that is no longer in use by the program and reclaims it. This frees the programmer from the burden of manual memory deallocation.

Question 259: What is exception handling?

Theory
Exception handling is a mechanism for responding to unexpected or exceptional conditions that arise during program execution. It allows a program to handle errors gracefully without crashing, typically using `try-except-finally` blocks.

Question 260: What is multithreading?

Theory
Multithreading is a model of program execution that allows for multiple threads (smaller units of a process) to be created within a single process. These threads can run concurrently, sharing the process's resources but executing independently. It is a way to achieve concurrency.

Question 261: What is concurrency?

Theory
Concurrency is the ability of different parts or units of a program to be executed out-of-order or in partial order, without affecting the final outcome. It is about **dealing with** lots of things at once. It does not necessarily mean they are running at the same instant in time (that is parallelism).

Question 262: What is a design pattern?

Theory
A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is a description or template for how to solve a problem, not a finished piece of code that can be transformed directly into code.

Question 263: What is modularity?

Theory
Modularity is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each module contains everything necessary to execute only one aspect of the desired functionality.

Question 264: What is coupling?

Theory
Coupling is the measure of the degree of interdependence between software modules. **Low coupling** is the goal, as it means that a change in one module will not require a change in another.

Question 265: What is cohesion?

Theory
Cohesion is the measure of the degree to which the elements inside a module belong together. **High cohesion** is the goal, as it means the module is focused on a single, well-defined task.

Question 266: What is the software development lifecycle?

Theory
The Software Development Lifecycle (SDLC) is a process used by the software industry to design, develop, and test high-quality software. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

Typical phases: Planning, Requirements, Design, Development, Testing, Deployment, Maintenance.

Question 267: What is version control?

Theory
Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows multiple developers to collaborate on a project without overwriting each other's work. Git is the most popular version control system.

Question 268: What is code review?

Theory
Code review is a software quality assurance activity in which one or several people check a program mainly by viewing and reading parts of its source code. It is a systematic examination of computer source code intended to find and fix mistakes overlooked in the initial development phase, improving both the quality of the software and the skills of the developers.

Of course. Here is a comprehensive list of answers to the Programming Basics interview questions, formatted for clarity and interview-readiness.

Algorithms & Searching/Sorting (20 Questions)

Problem-Solving Techniques

Question 269: What is binary search?

Theory
Binary search is a highly efficient searching algorithm. It works by repeatedly dividing the search interval in half. Its key prerequisite is that the data structure (e.g., an array) must be **sorted**.

How it Works:
1. Compare the target value with the middle element of the array.
2. If they match, the search is successful.
3. If the target is less than the middle element, continue the search on the **left half** of the array.
4. If the target is greater than the middle element, continue the search on the **right half**.

5. Repeat this process until the value is found or the interval is empty.
   Time Complexity: `O(log n)`.

Question 270: What is linear search?

Theory
Linear search is the simplest searching algorithm. It sequentially checks each element of a list until a match is found or the whole list has been searched. It does not require the list to be sorted.

Time Complexity: `O(n)`.

Question 271: What is sorting?

Theory
Sorting is the process of arranging the elements of a collection (like an array) in a specific order, either ascending or descending.

Question 272: What is bubble sort?

Theory
Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. It is highly inefficient.

Time Complexity: `O(n²)`.

Question 273: What is selection sort?

Theory
Selection sort is an in-place comparison sorting algorithm. It divides the list into a sorted and an unsorted sublist. It repeatedly finds the minimum element from the unsorted sublist and moves it to the end of the sorted sublist.

Time Complexity: `O(n²)`.

Question 274: What is insertion sort?

Theory
Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it is efficient for small datasets and is adaptive (efficient for data that is already substantially sorted).

Time Complexity: $O(n^2)$.

Question 275: What is merge sort?

Theory
Merge sort is an efficient, comparison-based sorting algorithm. It is a **divide and conquer** algorithm. It works by:

1. **Divide**: Recursively dividing the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
2. **Conquer**: Repeatedly merging the sublists to produce new sorted sublists until there is only one sorted list remaining.

Time Complexity: $O(n \log n)$.
Space Complexity: $O(n)$ (as it requires an auxiliary array for merging).

Question 276: What is quick sort?

Theory
Quick sort is another efficient, comparison-based sorting algorithm and is also a **divide and conquer** algorithm. It works by:

1. **Partition**: Picking an element as a **pivot** and partitioning the array around the pivot. All elements smaller than the pivot are moved to its left, and all elements greater are moved to its right.
2. **Recurse**: Recursively applying the above steps to the sub-arrays of elements with smaller and greater values.

Time Complexity: Average case is $O(n \log n)$, worst case is $O(n^2)$.
Space Complexity: $O(\log n)$ (due to recursion stack).

Question 280: What is Big O notation?

Theory
Big O notation is a mathematical notation used in computer science to describe the **asymptotic behavior** of a function's growth rate. It is used to classify algorithms according to how their run time (**time complexity**) or space requirements (**space complexity**) grow as the input size `n` grows. It describes the **worst-case scenario**.

Question 281: What is the difference between best case and worst case?

Theory
In algorithmic analysis:

- **Best Case**: The scenario that allows an algorithm to complete its execution in the minimum amount of time. It represents the lower bound on the algorithm's runtime.
- **Worst Case**: The scenario that causes an algorithm to complete its execution in the maximum amount of time. It represents the upper bound and is the most important for analysis (described by Big O).
- **Average Case**: The expected runtime of the algorithm, averaged over all possible inputs.

Question 282: What is the Fibonacci sequence?

Theory
The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1.
`0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

Question 283: How do you implement Fibonacci using recursion?

Code Example (Python)

```python
def fibonacci(n):
    # Base cases
    if n <= 1:
        return n
```

```
    else:
        # Recursive step
        return fibonacci(n-1) + fibonacci(n-2)
```

Note: This naive recursive implementation is highly inefficient (`O(2^n)`) due to repeated calculations. A more efficient approach would use dynamic programming (memoization).

Question 284: What is dynamic programming?

Theory
Dynamic programming is an optimization technique for solving complex problems by breaking them down into simpler subproblems. It solves each subproblem only once and stores its solution, typically in a table (memoization). When the same subproblem is encountered again, it retrieves the stored solution instead of recomputing it.

Question 285: What is a greedy algorithm?

Theory
A greedy algorithm is an algorithmic paradigm that makes the **locally optimal choice** at each stage with the hope of finding a global optimum. It makes the best choice at the current moment without regard for the future.

Question 286: What is divide and conquer?

Theory
Divide and conquer is an algorithm design paradigm. It works by recursively breaking down a problem into two or more subproblems of the same or related type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.
Examples: Merge sort, Quick sort.

Question 287: What is breadth-first search?

Theory

Breadth-First Search (BFS) is a graph traversal algorithm that starts at the root node and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It uses a **queue** data structure.

Question 288: What is depth-first search?

Theory
Depth-First Search (DFS) is a graph traversal algorithm that starts at the root node and explores as far as possible along each branch before backtracking. It uses a **stack** data structure (often the call stack, via recursion).

Practical Programming (15 Questions)

Coding Problem Solutions

Question 289: How do you reverse a string?

Python Solution

```
s = "hello"
reversed_s = s[::-1] # Using slicing
```

Question 290: How do you check if a string is a palindrome?

Python Solution

```
def is_palindrome(s):
    # Normalize the string (lowercase, remove non-alphanumeric)
    s = ''.join(char.lower() for char in s if char.isalnum())
    return s == s[::-1]
```

Question 291: How do you find the largest element in an array?

Python Solution

```
arr = [3, 5, 1, 9, 2]
max_element = max(arr)
```

Question 292: How do you find duplicates in an array?

Python Solution

```
from collections import Counter

arr = [1, 2, 3, 2, 4, 3, 5]
counts = Counter(arr)
duplicates = [item for item, count in counts.items() if count > 1]
# duplicates will be [2, 3]
```

Question 293: How do you remove duplicates from an array?

Python Solution

```
arr = [1, 2, 3, 2, 4, 3, 5]
unique_list = list(set(arr))
```

Question 297: How do you find the factorial of a number?

Python Solution (Recursive)

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Question 298: How do you check if a number is prime?

Python Solution

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Question 300: How do you find the GCD of two numbers?

Python Solution (Euclidean algorithm)

```python
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

Question 303: How do you implement binary search?

Python Solution (Iterative)

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid # Found
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1 # Not found
```