**1) What is risk-based testing, and how does it apply to advanced projects?**

* A testing approach that **prioritizes test efforts** based on the risk of failure and its impact.
* Focuses on **critical functionalities, business-impact areas, or frequently failing modules**.
* In advanced projects, it helps **optimize resources**, reduce time, and focus on areas with **maximum risk exposure**.

**Scenario:** A banking application has modules for fund transfer, account statement, and promotional offers.  
**Application:** Focus testing on **fund transfer** first because it's high risk (money involved), while promotions are low risk and can be deprioritized.

**2) How do you design test cases for highly complex applications?**

* Understand **business logic, system architecture, and dependencies**.
* Break down features into **functional modules**.
* Use **equivalence partitioning, boundary value analysis, and state transition** techniques.
* Prioritize **integration, data flow, edge case** testing.
* Include **both positive and negative scenarios**.

**Scenario:** Testing an airline reservation system.  
**Approach:**

* **Divide**: Booking, Payment, Cancellation, Check-in.
* For Booking, create cases like:
  + Book one-way trip with different class options.
  + Book round trip with invalid dates.
  + Book with special discount codes (edge cases).

**3) What is fault seeding in software testing?**

* A technique where **known defects are intentionally injected** into the system.
* Used to **evaluate the effectiveness** of the testing process and team.
* If seeded defects are found, it indicates a **strong detection process**; if not, the process may need improvement.

**Scenario:** Insert a known bug like allowing login with an expired password.  
**Use:** If testers detect this intentionally seeded bug, it means the **testing approach is effective**.

**4) How do you handle test case prioritization in large-scale projects?**

* Prioritize based on:
  + **Business impact**
  + **Failure probability**
  + **Complexity and criticality**
  + **Usage frequency**
* Categories:
  + **High Priority** – Core workflows
  + **Medium** – Secondary features
  + **Low** – UI, cosmetic or rarely used features

**Scenario:** E-commerce app with modules: Login, Cart, Checkout, Wishlist.  
**Application:**

* High priority: **Checkout and Payment**
* Medium: **Cart & Login**
* Low: **Wishlist**

**5) What is model-based testing, and how is it applied?**

* A testing technique where **test cases are derived from models** representing system behavior (e.g., state machines, flowcharts).
* Applied by:
  + Creating a **functional or behavioral model**
  + Generating test cases automatically from the model
  + Validating whether the system behaves according to the model
* Useful for **complex, rule-driven applications**.

**Scenario:** ATM machine  
**Model:** State diagram with states like Card Inserted → PIN Entered → Transaction Selected → Cash Dispensed.  
**Use:** Derive test cases like:

* Insert card, enter wrong PIN 3 times → Block card
* Enter correct PIN → Withdraw cash

**6) How do you deal with dynamic requirements in manual testing?**

* Stay **closely aligned with the development team** and product owner.
* Use **exploratory testing** to handle on-the-fly changes.
* Maintain a **traceability matrix** to track which test cases are impacted.
* Frequently **review and update** test cases based on the latest requirements.

**Scenario:** Login page suddenly requires a CAPTCHA in the next sprint.  
**Action:**

* Update test cases
* Create exploratory tests for CAPTCHA validation
* Add cases for failed CAPTCHA attempts, timeouts, etc.

**7) What is mutation testing, and how is it performed?**

* Involves modifying code slightly (e.g., changing a condition) to create **mutants**.
* Existing test cases are run to **"kill" these mutants**.
* If mutants survive, the test cases need improvement.
* Typically used in **white-box or automation** environments, but the concept helps enhance **manual test design** too.

**8) What are the challenges in performing security testing manually?**

* Requires **deep knowledge of security vulnerabilities** (e.g., XSS, SQL Injection).
* Some attacks are **difficult to simulate manually**.
* Time-consuming to cover **all security entry points**.
* Tools like Burp Suite or OWASP ZAP are often needed even in partially manual testing.

**Scenario:** Testing a login form for SQL Injection.  
**Attempt:** Enter admin' -- in username.  
**Challenge:** Without tools, identifying where data flows and whether it's sanitized is **hard and time-consuming**.

**9) How do you ensure full test coverage in a multi-tier application?**

* Test across **all layers**: UI, business logic, database, APIs.
* Use **Requirement Traceability Matrix (RTM)** to map all requirements to test cases.
* Include **integration, system, E2E, and data flow** scenarios.
* Collaborate with developers for **gray-box** insights.

**Scenario:** Online shopping app with Web (UI), API, and Database.  
**Approach:**

* UI: Add-to-cart functionality
* API: Validate cart API with Postman
* DB: Check if cart items reflect in the database  
  **Tools:** RTM ensures every requirement has test cases mapped.

**10) What is gray-box testing, and when should it be used?**

* A hybrid testing method combining **white-box (code-level) and black-box (functionality-level)** approaches.
* Testers have **partial knowledge** of internal workings (e.g., database, architecture).
* Ideal for testing **complex workflows, data validation, and integration scenarios**.

**Scenario:** Testing student marks calculation in a school management system.  
**Gray-box insight:** You know the formula (e.g., 30% internal + 70% external) but don't see code.  
**Test:**

* Enter internal=18, external=60 → Expect 180.3 + 600.7 = 50.4
* Use that partial internal knowledge to validate logic.

**11) How do you perform compatibility testing for multiple devices and browsers?**

* Test on a **matrix of OS-browser-device combinations** (e.g., Windows + Chrome, Android + Firefox).
* Use tools like **BrowserStack** or real devices.
* Validate **UI rendering, functionality, and responsiveness** across platforms.
* Prioritize based on **user traffic analytics**.

**Scenario:** E-commerce app’s “Add to Cart” button looks perfect on Chrome (Windows), but **disappears on Safari (iOS)**.  
**Action:** Test the site on **multiple browsers/devices using BrowserStack**, note down rendering/functionality issues, and report them.

**12) What is the importance of orthogonal array testing?**

* A **systematic technique** to reduce the number of test combinations.
* Ensures **maximum coverage with minimum tests**.
* Useful when testing **multiple input combinations** in areas like configuration testing.
* Saves time while maintaining effectiveness.

**Scenario:** Mobile recharge system with 3 operators, 3 recharge types, and 3 payment methods = 27 combinations.  
**Solution:** Use **Orthogonal Array** to test just **9 optimized combinations** that ensure each input type is tested at least once.

**13) What is the difference between fault tolerance and fault recovery testing?**

* **Fault Tolerance**: System's ability to **continue functioning** during failures.
  + *E.g.*, a failover server taking over.
* **Fault Recovery**: System’s ability to **recover after a failure**.
  + *E.g.*, auto-restart or re-try mechanisms after crash.

**Fault Tolerance:** If the database fails, the app switches to a **replica server** and continues functioning.

**Fault Recovery:** If the app crashes during payment, a **recovery process auto-resumes** from the last checkpoint when restarted.

**14) How do you validate data integrity manually?**

* Compare **source and target data** after operations like migration, ETL, or CRUD operations.
* Use **SQL queries** to validate consistency, completeness, and accuracy.
* Check for **data loss, corruption, duplication**, and truncation.

**Scenario:** After user registration, verify the user data in the database.  
**Action:** Manually run SQL: SELECT \* FROM users WHERE email = 'test@example.com';

Check if all values are stored accurately (name, date of birth, phone).

**15) What is equivalence class partitioning, and how does it improve test design?**

* A **black-box test technique** that divides inputs into **valid and invalid partitions**.
* From each class, select **one representative value**.
* Reduces the number of test cases while **ensuring coverage**.
  + *E.g.*, Input field accepts 1–100 → Test 50 (valid), 0 & 101 (invalid).

**16) How do you conduct data-driven testing without automation?**

* Use **manual test cases with varied input data** sets from Excel, CSV, or SQL tables.
* Run the same test steps with **different inputs manually**.
* Track expected vs. actual results in test logs or sheets.

**Scenario:** Login functionality needs to be tested with 5 user credentials.  
**Action:**

* Keep credentials in Excel
* Execute the same test steps manually for each set
* Log results per user in the sheet

**17) What is risk mitigation in software testing?**

* Actions taken to **reduce the impact or probability** of risks.
* Includes **early testing, test coverage mapping, prioritizing critical modules**, and backups.
* Helps prevent **major failures post-release**.

**Scenario:** Banking app has high-risk transaction modules.  
**Mitigation Steps:**

* Assign senior testers
* Begin testing early
* Add smoke tests
* Include security validations

**18) What is fuzz testing, and how is it performed manually?**

* Inputting **unexpected, random, or malformed data** to a system to check its robustness.
* Manual approach: Try **nonsensical strings, long inputs, special characters** in input fields.
* Observe if the application **crashes, hangs, or behaves unexpectedly**.