UNIT–4 : Software Testing (PYQ Solution)

AKTU · Software Engineering (BCS601) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. Define Pareto Principle. (May 2026)

The Pareto Principle (80-20 rule) states that roughly 80% of effects come from 20% of causes. In software testing and quality, about 80% of the defects are found in about 20% of the modules. So the testers first identify these "vital few" error-prone modules and concentrate testing and corrective effort on them, giving the greatest quality improvement for the least effort.

Q2. Discuss following type of testing techniques (i) Mutation Testing (ii) Regression Testing (May 2026)

(i) Mutation testing is a fault-based technique in which small changes (mutants), such as changing + to –, are introduced into the program. Test cases are run on each mutant: if a test fails, the mutant is killed; if not, the tests are weak. The mutation score measures test-suite quality. (ii) Regression testing re-runs existing tests after a modification to confirm that the change has not broken earlier working features. It is repeated after every fix or enhancement, often automated.

Q3. Justify the statement "Exhaustive Testing is not possible". (May 2025)

Exhaustive testing means testing with all possible inputs and all paths. Even a simple program has far too many combinations: a function with two 32-bit integers has 2^{64} input pairs, and a loop that may execute many times creates an enormous number of paths. Testing them all would take years and unlimited cost. Hence testers select representative cases using equivalence partitioning, boundary value analysis and path coverage, and testing can only show the presence of errors, never their absence.

Q4. Distinguish between verification and validation. (2023)

Verification asks "Are we building the product right?" It checks that each phase output conforms to its specification through reviews, walkthroughs and inspections; it is static and done by the QA team. Validation asks "Are we building the right product?" It checks that the final software meets the user's actual needs by executing it through testing; it is dynamic and done on the running product, often with the customer.

Q5. Distinguish between alpha and beta testing? (2023)

Alpha testing is done at the developer's site by internal staff or selected users in a controlled environment, and the developer records defects immediately. Beta testing is done at the customer's site by real end users in the live environment, without the developer present; users report problems back. Alpha comes first and finds most bugs; beta follows and gathers feedback on real-world use before final release.



SECTION B Answers (≈250 Words Each)

Q1. Define the following terms (i) Fault (ii) Failure (iii) Mistake (iv) Bug (May 2026)

These are related terms describing how an error travels from a person to a visible problem.

- **(i) Fault (defect):** an incorrect step, instruction or data definition in the program, which is the result of a mistake. It may stay hidden until the code is executed under particular conditions. Example: using < instead of <= in a loop condition.
- **(ii) Failure:** the **visible deviation** of the program's actual behaviour from the expected behaviour, which appears when a fault is executed. Example: the loop skips the last element and the report shows a wrong total.
- **(iii) Mistake (error):** a **human action** that produces an incorrect result, such as misunderstanding a requirement or a typing slip by the programmer. Mistakes cause faults.
- **(iv) Bug:** the informal name for a fault or defect found in the code during testing or use; the programmer then **debugs** the program to remove it.

Chain: Mistake → Fault (bug) → Failure.

Term	Where it is found	Who causes it
Mistake	In the mind/action of developer	Developer
Fault/Bug	In the code or design	Result of mistake
Failure	In execution output	Seen by tester/user

Conclusion: a mistake by a person introduces a fault (bug) into the software, and the fault shows up as a failure when the faulty code runs.

Q2. Discuss following type of testing techniques in details. 1. Unit Testing 2. Integration Testing 3. System Testing (May 2025)

These are the main **levels of testing**, performed in sequence.

1. Unit testing

- Tests the smallest unit (function, class or module) in isolation, using **white-box** techniques by the developer.
- Checks interface, local data, boundary conditions, independent paths and error handling.
- Uses **drivers** (call the unit) and **stubs** (replace called modules).
- Finds most coding errors early, cheaply.

2. Integration testing

- Combines tested units and tests their **interfaces and interactions**.
- Approaches: **big-bang** (all at once), **top-down** (with stubs), **bottom-up** (with drivers), **sandwich/hybrid**.
- Finds interface mismatch, wrong data passing and timing errors.

3. System testing

- Tests the **complete, integrated system** against the SRS using **black-box** techniques by an independent test team.
- Types: **functional, performance, stress, security, recovery, usability, compatibility** testing.
- Done in an environment close to production.



Levels of testing

Level	Target	Method	Tester
Unit	Single module	White box	Developer
Integration	Module interfaces	Both	Developer/tester
System	Whole system	Black box	Test team

Conclusion: unit testing finds module defects, integration testing finds interface defects, and system testing confirms that the whole product meets its requirements.

Q3. Explain boundary value analysis and its significance with example. (2023)

Boundary Value Analysis (BVA) is a **black-box** technique that selects test cases at the **edges of input ranges**, because programmers often make errors at boundaries (such as using < instead of <=).

Procedure: for a variable in the range [min, max], test **min, min+1, nominal (typical value), max−1 and max**. For n variables, vary one at a time and hold the others at nominal, giving **4n + 1** test cases. **Robust BVA** adds min−1 and max+1 (6n + 1 cases); **worst-case BVA** takes all combinations (5^n cases).

Example: a field accepts age from 18 to 60.

Test	Value	Expected
min−1 (robust)	17	Rejected
min	18	Accepted
min+1	19	Accepted
nominal	40	Accepted
max−1	59	Accepted
max	60	Accepted
max+1 (robust)	61	Rejected

Significance

- Finds boundary defects, which are the commonest input errors.
- Gives a small, systematic test set for a large input range.
- Needs no knowledge of the code.
- Works well together with **equivalence partitioning**.

Limitations: it does not test logical dependencies between variables and suits only ordered numeric inputs.

Conclusion: BVA gives high defect detection with few test cases and is a key black-box technique.

SECTION C Answers (≈500 Words Each)

Q1. What do you understand by the Program Verification and Validation? Discuss about Static and Dynamic Analyzers. (May 2026)

Verification checks that the software correctly implements a specification at each step: "Are we building the product right?" It uses reviews, walkthroughs, inspections and static analysis, and does not need the program to run. **Validation** checks that the finished software meets the user's real needs: "Are we building the right product?" It uses testing (unit, integration, system, acceptance) on the running program.

Point	Verification	Validation
Question	Right product built correctly?	Right product built?
Method	Static: review, inspection	Dynamic: testing
Object	Documents, design, code	Executable software
Done by	QA team	Testers and customer
Timing	Before and during development	After code is ready

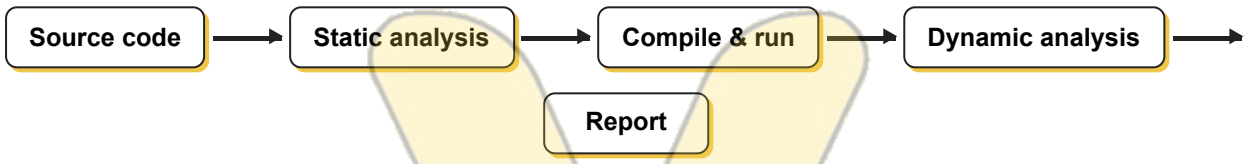
Together they are called **V&V** and are performed through the whole life cycle to find defects early and gain confidence in quality.

Static analyzers examine the program **without executing it**, working on source code or design.

- **Control-flow analysis** – finds unreachable code, infinite loops and loops with multiple exits.
- **Data-use analysis** – finds variables used before assignment, declared but never used, or assigned twice without use.
- **Interface analysis** – checks consistency of procedure and function calls with their declarations.
- **Information-flow and path analysis** – shows dependencies between inputs and outputs and the paths through the code.
- Also report coding-standard violations, complexity metrics and security weaknesses (e.g. lint, SonarQube, FindBugs).
- **Advantages:** early detection and low cost; covers all paths. **Limits:** false warnings; cannot find run-time behaviour problems.

Dynamic analyzers work **by executing the program** with test data and observing its behaviour.

- **Test-coverage tools** – measure which statements, branches and paths are executed by the tests.
- **Profilers** – measure time and memory used by each function, finding performance bottlenecks.
- **Memory analyzers** – detect memory leaks, buffer overflows and invalid pointers (e.g. Valgrind).
- **Tracers / debuggers and assertion checkers** – log execution paths and check conditions at run time.
- **Advantages:** find real run-time errors and performance problems. **Limits:** only cover the paths actually run; need test cases.



Static analysis before execution, dynamic analysis during execution

Conclusion: verification and validation together assure quality; static analyzers find defects without running the code and dynamic analyzers find those that appear only at run time.

Q2. Draw the Control Flow Graph (CFG) and calculate cyclomatic complexity of the following procedure. void sort (int a[], int n) { int i=0, j; while (i<n-1) { j=i+1; while (j<n) { if (a[i]>a[j]) { temp=a[i]; a[i]=a[j]; a[j]=temp; } j=j+1; } i=i+1; } } (May 2026)

A **Control Flow Graph** has a node for each statement (or block) and an edge for each possible transfer of control. **Cyclomatic complexity V(G)** (McCabe) measures the number of independent paths.

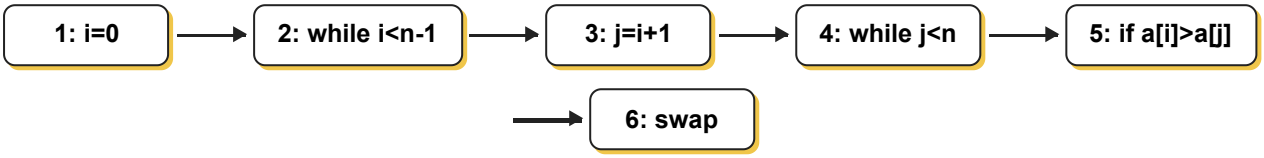
Numbering the nodes

1. `i = 0` (initialisation)
2. `while (i < n−1)` (decision)
3. `j = i + 1`
4. `while (j < n)` (decision)
5. `if (a[i] > a[j])` (decision)

6. `temp = a[i]; a[i] = a[j]; a[j] = temp;` (swap block)
7. `j = j + 1`
8. `i = i + 1`
9. End

Edges

1→2, 2→3 (true), 2→9 (false), 3→4, 4→5 (true), 4→8 (false), 5→6 (true), 5→7 (false), 6→7, 7→4, 8→2.



CFG main path (7: j=j+1 loops back to 4; 4 false goes to 8: i=i+1 then back to 2; 2 false goes to 9: end)

Cyclomatic complexity

- Number of nodes N = 9, edges E = 11.
- **V(G) = E − N + 2 = 11 − 9 + 2 = 4**
- **V(G) = P + 1** where P = predicate nodes (2, 4, 5) = 3, so V(G) = 3 + 1 = **4**.
- **V(G) = number of regions** = 4 (inner if region, inner loop region, outer loop region, outside region).

Independent (basis) paths

1. 1-2-9 (outer loop never entered)
2. 1-2-3-4-8-2-9 (inner loop not entered)
3. 1-2-3-4-5-7-4-8-2-9 (condition false, no swap)
4. 1-2-3-4-5-6-7-4-8-2-9 (swap executed)

Interpretation: V(G) = 4 means at least four test cases are needed to execute every independent path; as 4 is below 10, the procedure is simple and low-risk.

Conclusion: the CFG of the sort procedure has 9 nodes and 11 edges, and its cyclomatic complexity is 4.

Q3. Consider a program to classify a Triangle. Its input is triple of positive integers (X,Y,Z) and data type of input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following: Scalene, Isosceles,

Equilateral, Not a Triangle. Design the Boundary Value Test Cases. (May 2026)

Boundary Value Analysis tests each variable at its extremes. For X, Y, Z in the range [1, 100], the boundary values are **min = 1, min+ = 2, nominal = 50, max− = 99, max = 100**. In normal BVA one variable is varied through these five values while the others stay at the nominal value, giving **4n + 1 = 4 × 3 + 1 = 13 test cases**.

Rules: the sides form a triangle only if the sum of any two sides is greater than the third. Equilateral: all three equal; isosceles: exactly two equal; scalene: all different; otherwise "Not a Triangle".

Case	X	Y	Z	Expected output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a Triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a Triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a Triangle

Check of an important case: (50, 50, 100): 50 + 50 = 100, which is not greater than 100, so it is **Not a Triangle**. (50, 50, 99): 50 + 50 = 100 > 99, so it is a valid isosceles triangle.

Variants

- **Robust BVA** adds min−1 = 0 and max+1 = 101 for each variable (6n + 1 = 19 cases); these would be rejected as invalid input.

- **Worst-case BVA** takes all combinations of the five values: 5^3 = **125 test cases**, which would also include scalene cases such as (2, 99, 100) and (99, 100, 100).

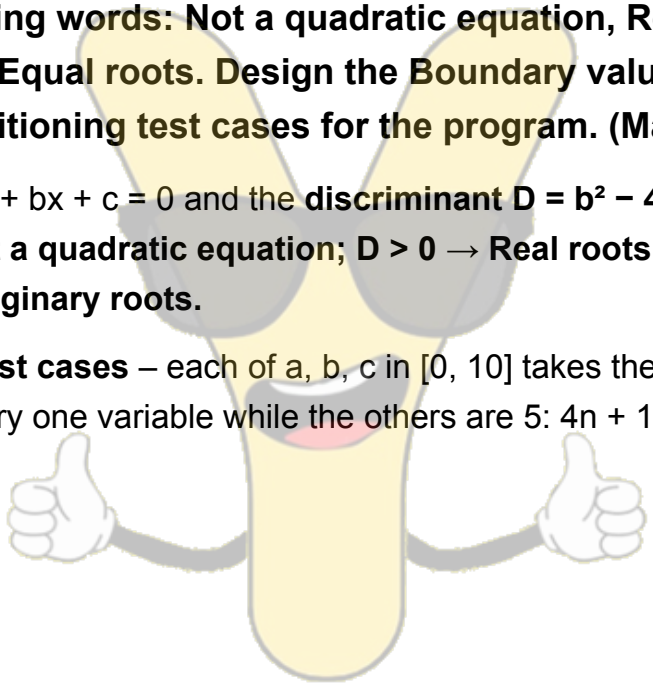
Because the 13 cases never give a **Scalene** result, a few extra cases from equivalence classes are advisable, e.g. (3, 4, 5) Scalene.

Conclusion: normal BVA gives 13 test cases that exercise the lower and upper boundaries of each side and detect the typical off-by-one errors in the triangle inequality.

Q4. Consider a program for determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values may be from interval [0, 10]. The program output may have one of the following words: Not a quadratic equation, Real roots, Imaginary roots, Equal roots. Design the Boundary value and Equivalence Partitioning test cases for the program. (May 2025)

The equation is $ax^2 + bx + c = 0$ and the **discriminant $D = b^2 - 4ac$** decides the output: **$a = 0$ → Not a quadratic equation; $D > 0$ → Real roots; $D = 0$ → Equal roots; $D < 0$ → Imaginary roots.**

Boundary Value test cases – each of a, b, c in [0, 10] takes the values 0, 1, 5 (nominal), 9, 10. Vary one variable while the others are 5: 4n + 1 = **13 cases**.



Case	a	b	c	D	Expected output
1	0	5	5	25	Not a quadratic equation
2	1	5	5	5	Real roots
3	5	5	5	−75	Imaginary roots
4	9	5	5	−155	Imaginary roots
5	10	5	5	−175	Imaginary roots
6	5	0	5	−100	Imaginary roots
7	5	1	5	−99	Imaginary roots
8	5	9	5	−19	Imaginary roots
9	5	10	5	0	Equal roots
10	5	5	0	25	Real roots
11	5	5	1	5	Real roots
12	5	5	9	−155	Imaginary roots
13	5	5	10	−175	Imaginary roots

(Robust BVA would add the invalid values −1 and 11 for each variable, which must be rejected.)

Equivalence partitioning – divide the input into classes where the program behaves in the same way, and test one value from each class.

Class	Condition	Test (a, b, c)	Expected output
V1	a = 0	(0, 3, 4)	Not a quadratic equation
V2	a > 0 and D > 0	(1, 5, 2) → D = 17	Real roots
V3	a > 0 and D = 0	(1, 2, 1) → D = 0	Equal roots
V4	a > 0 and D < 0	(1, 1, 1) → D = −3	Imaginary roots
I1	a, b or c < 0	(−1, 2, 3)	Invalid input
I2	a, b or c > 10	(11, 2, 3)	Invalid input

Conclusion: the 13 boundary cases test the edges of the input range and the 4 valid and 2 invalid equivalence classes cover each type of output with very few tests.

Q5. Draw the Control Flow Graph (CFG) for the following program and calculate its Cyclomatic Complexity. Also write various independent/basis paths available in the program/CFG. int compute_gcd(x,y) int x,y; { while (x != y) { if (x>y) then x=x-y; else y=y-x; } return x; } (May 2025)

Nodes

1. `while (x != y)` (decision)
2. `if (x > y)` (decision)
3. `x = x − y`
4. `y = y − x`
5. `return x` (end)

(The entry is the while node, since no statement precedes it.)

Edges: 1→2 (true), 1→5 (false), 2→3 (true), 2→4 (false), 3→1, 4→1.



CFG: 1 to 2; 2 to 3 (true) or 4: y=y-x (false), both return to 1; 1 to 5: return x (false)

Cyclomatic complexity

- N = 5, E = 6, so **V(G) = E − N + 2 = 6 − 5 + 2 = 3**.
- Predicates P = 2 (nodes 1 and 2): **V(G) = P + 1 = 3**.
- Regions: two bounded regions (the two loop paths) plus the outer region = **3**.

Independent (basis) paths

1. **1** → **5** (x = y at the start; returns x)
2. **1** → **2** → **3** → **1** → **5** (x > y once, then x = y)
3. **1** → **2** → **4** → **1** → **5** (x < y once, then x = y)

Test cases for each path

Path	Input (x, y)	Result
1	(5, 5)	5
2	(10, 5)	5
3	(5, 10)	5

Interpretation: V(G) = 3 is the minimum number of test cases required to execute every basis path; the function is simple, as the value is below 10.

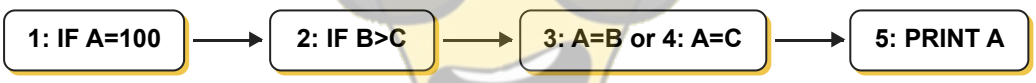
Conclusion: the GCD program has 5 nodes and 6 edges, giving a cyclomatic complexity of 3 and three basis paths.

Q6. Draw the control flow graph and Calculate the cyclomatic complexity with three methods and independent paths for the given code: IF A = 100 THEN IF B > C THEN A = B ELSE A = C ENDIF ENDIF PRINT A (2023)

Nodes

1. `IF A = 100` (decision)
2. `IF B > C` (decision)
3. `A = B`
4. `A = C`
5. `PRINT A` (end)

Edges: 1→2 (A = 100 true), 1→5 (false), 2→3 (true), 2→4 (false), 3→5, 4→5. So **N = 5, E = 6.**



CFG: 1 true to 2, 1 false to 5; 2 true to 3, false to 4; 3 and 4 to 5

Three methods of cyclomatic complexity

1. **Edges and nodes:** V(G) = E − N + 2 = 6 − 5 + 2 = **3**.
2. **Predicate nodes:** V(G) = P + 1 = 2 + 1 = **3** (nodes 1 and 2).
3. **Regions:** the graph has 2 bounded regions plus 1 outer region, so V(G) = **3**.

Independent paths (three, equal to V(G))

1. 1 → 5 (A ≠ 100)
2. 1 → 2 → 3 → 5 (A = 100 and B > C)
3. 1 → 2 → 4 → 5 (A = 100 and B ≤ C)

Test cases

Path	A	B	C	Output
1	50	any	any	A = 50
2	100	8	3	A = 8
3	100	2	6	A = 6

Conclusion: the code has cyclomatic complexity 3, i.e. three independent paths and a minimum of three test cases.

Q7. Discuss the various types of structural testing techniques with example of each. (2023)

Structural (white-box) testing designs test cases from the **internal structure of the code**. Its techniques are based on coverage criteria.

Example code used: `if (x > 0 && y > 0) z = x + y; else z = 0;`

- **Statement coverage** – every statement executes at least once. Test (x = 1, y = 1) executes `z = x + y`, and (x = −1, y = 1) executes `z = 0`; two tests give 100%.
- **Branch (decision) coverage** – every decision takes both true and false outcomes. The same two tests cover both branches.
- **Condition coverage** – each atomic condition is made true and false. Tests (x = 1, y = −1) and (x = −1, y = 1) do this.
- **Decision/condition and multiple-condition coverage** – all combinations of condition outcomes (TT, TF, FT, FF), giving 2^n tests for n conditions.
- **Path coverage** – every independent path executes; **cyclomatic complexity V(G)** gives the number of basis paths (here V(G) = 3).
- **Loop testing** – skip the loop, one pass, two passes, typical n, maximum, maximum+1 passes.
- **Data-flow testing** – tests definition-use pairs of variables, e.g. the path from the assignment `z = x + y` to the later use of z.
- **Mutation testing** – small faults (mutants) are inserted to see whether tests detect them.

Technique	Covers	Strength
Statement	All statements	Weakest
Branch	All decisions both ways	Stronger
Condition	Each condition	Stronger
Path	All independent paths	Strong
Data flow	Def-use pairs	Finds data errors



Structural coverage criteria in increasing strength

Conclusion: structural testing derives tests from code structure; stronger criteria find more defects but need more test cases.

