Notes Link

Software Testing & Quality Assurance

Practice Set

Target Group : 3rd Year B.Tech

1. Explain Alpha Testing, Beta Testing, Unit Testing, Integration Testing, System Testing, Regression Testing.

Testing Type	Description
Alpha Testing	A type of software testing performed to identify bugs before releasing the product to real users or to the public. It is done early on, near the end of the development of the software, typically by in-house software engineers or quality assurance staff.
Beta Testing	The process of testing a software product or service in a real-world environment before its official release. It is an essential step in the software development lifecycle as it helps identify bugs and errors that may have been missed during the development process.
Unit Testing	A type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements.
Integration Testing	The process of testing the interface between two software units or modules. It focuses on determining the correctness of the interface. The purpose of integration testing is to expose faults in the interaction between integrated units.
System Testing	A type of software testing that evaluates the functionality and performance of a complete and integrated software solution. It tests if the system meets the requirements and is suitable for delivery to the end-users.
Regression Testing	A software testing process executed after making modifications or upgrades to a software application and re-tests the application areas that may have been affected by the fix.

2. Explain White Box Testing with its applications, challenges, merits and demerits.

Aspect	Description
White Box Testing	White Box Testing, also known as clear box testing, glass box testing, transparent box testing, and structural testing, is a method of software testing that tests internal structures or workings of an application[1234]. The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level[1].
Applications	White Box Testing is used to analyze the internal structures, the used data structures, internal design, code structure, and the working of the software[1]. It is used to test the software's internal logic, flow, and structure[1]. The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements[1].
Challenges	Some of the challenges faced in White Box Testing include a lack of understanding of the programming language used for testing, a lack of understanding of the logical flow/use case, a lack of patience to go through the program, the reality of testing, copying the existing functionality, allowing clients/managers to influence, and not being honest enough with the client[5].
Merits	The merits of White Box Testing include better code coverage, detection of defects and potential issues early on in the development process, improved code quality, improved security, and allowance for timely remediation[678].
Demerits	The demerits of White Box Testing include high cost, frequently changing code, missing cases, and it becomes time-consuming

3. Explain Black Box Testing with its applications, challenges, merits and demerits.

Aspect	Description
Black Box Testing	Black Box Testing is a software testing method that examines the functionality of an application without peering into its internal structures or workings[1234]. The tester provides an input, and observes the output generated by the system under test[1].
Applications	Black Box Testing is used to validate an app's functional requirements[1]. It can quickly identify how the system responds to expected and unexpected user actions, its response time, usability issues, and reliability issues[1]. It is applied to all software testing levels[1].
Challenges	Some challenges of Black Box Testing include the possibility of repeating the same tests while implementing the testing process[1], test cases being difficult to design

Aspect	Description
	without clear functional specifications[5], and it being difficult to execute the test cases because of complex inputs at different stages[5].
Merits	The merits of Black Box Testing include unbiased tests because the designer and tester work independently[5], the tester being free from any pressure of knowledge of specific programming languages to test the reliability and functionality of an application[5], and it facilitates identification of contradictions and vagueness in functional specifications[5].
Demerits	The demerits of Black Box Testing include tests can be redundant if already run by the software designer[5], test cases are extremely difficult to be designed without clear and concise specifications[5], and testing every possible input stream is not possible because it is time-consuming and this would eventually leave many program paths untested

4. Explain Debugging with its tools.

Aspect	Description
Debugging	Debugging is the process of identifying and resolving errors, or bugs, in a software system[123]. It is an important aspect of software engineering because bugs can cause a software system to malfunction, and can lead to poor performance or incorrect results[123]. Debugging can be a time-consuming and complex task, but it is essential for ensuring that a software system is functioning correctly[123].
Tools	There are various tools available for debugging such as debuggers, trace tools, and profilers that can be used to identify and resolve bugs[1]. Some of the widely used debuggers are Visual Studio Code, Android Studio, IntelliJ IDEA, ReSharper, PyCharm Debugger, Chrome DevTools, and X code[4]. These tools provide a specialized environment for controlling and monitoring the execution of a program[1].

Examples of automated debugging tools include code-based tracers, profilers, interpreters, etc. Some of the widely used debuggers are:

- Radare2
- WinDbg
- Valgrind

5. Explain Cyclomatic Complexity or Macabe's Path Method with examples.

Aspect	Description
Cyclomatic Complexity	Cyclomatic Complexity is a software metric used to measure the logical complexity of a program[1234]. It is a quantitative measure of the number of linearly independent paths in the source code of a software program[1234]. It was developed by Thomas J. McCabe in 1976[1234].
Applications	Cyclomatic Complexity is used to determine the complexity of a program. It counts the number of decisions in the source code[4]. The higher the count, the more complex the code[4]. It is used to validate an app's functional requirements[1]. It can quickly identify how the system responds to expected and unexpected user actions, its response time, usability issues, and reliability issues[1]. It is applied to all software testing levels[1].
Challenges	Some challenges of Cyclomatic Complexity include the possibility of repeating the same tests while implementing the testing process[1], test cases being difficult to design without clear functional specifications[1], and it being difficult to execute the test cases because of complex inputs at different stages[1].
Merits	The merits of Cyclomatic Complexity include unbiased tests because the designer and tester work independently[1], the tester being free from any pressure of knowledge of specific programming languages to test the reliability and functionality of an application[1], and it facilitates identification of contradictions and vagueness in functional specifications[1].
Demerits	The demerits of Cyclomatic Complexity include tests can be redundant if already run by the software designer[1], test cases are extremely difficult to be designed without clear and concise specifications[1], and testing every possible input stream is not possible because it is time-consuming and this would eventually leave many program paths untested[1].

Here's an example of how to calculate Cyclomatic Complexity:

```
A = 10
if B > C:
    A = B
else:
    A = C
print(A)
print(B)
print(C)
```

6. Explain Manual and Automation Testing.

Aspect	Manual Testing	Automation Testing
Definition	Manual testing is a software testing process in which test cases are executed manually without using any automated tool[1][2][3][4].	Automation Testing uses automation tools to execute test cases[1].
Processing Time	Manual testing is time-consuming and takes up human resources[1].	Automated testing is significantly faster than a manual approach[1].
Exploratory Testing	Exploratory testing is possible in Manual Testing[1].	Automation does not allow random testing[1].
Initial Investment	The initial investment in the Manual testing is comparatively lower[1].	The initial investment in the automated testing is higher. Though the ROI is better in the long run[1].
Pros	Manual testing is easy to implement, provides fast feedback, is versatile, flexible, and less expensive[1].	Automated testing is more reliable, quicker, allows for different, complex types of testing, improves project quality, and doesn't find visual or UX bugs[4].
Cons	In manual testing, not all defects are detected, high expertise is required, it takes a lot of time, cannot be recorded, and is less reliable[2].	In automated testing, the team can spend a lot of time debugging tests[4]

7. Explain Software Quality Assurance with examples.

Aspect	Description
Software Quality Assurance (SQA)	Software Quality Assurance (SQA) is a process that assures that all software engineering processes, methods, activities, and work items are monitored and comply with the defined standards[1][2]. These defined standards could be one or a combination of anything like ISO 9000, CMMI model, ISO15504, etc[2]. SQA incorporates all software development processes starting from defining requirements to coding until release[2]. Its prime goal is to ensure quality[2].
Applications	SQA is applied throughout the software process[1]. It ensures that standards that have been adopted are followed, and all work products conform to them[1]. SQA encompasses SQA process specific quality assurance and quality control tasks (including technical reviews and a multitiered testing strategy), effective software engineering practice (methods and tools), control of all software work products and the changes made to them, a procedure to ensure compliance with software development standards (when applicable), and measurement and reporting

Aspect	Description
Examples	mechanisms[1]. Here are some examples of SQA activities[2]: 1**Creating an SQA Management Plan**: Charting out a blueprint of how SQA will be carried out in the project with respect to the engineering activities while ensuring that you corral the right talent/team[2]. 2**Setting the Checkpoints**: The SQA team sets up periodic quality checkpoints to ensure that product development is on track and shaping up as expected[2]. 3**Support/Participate in the Software Engineering team's requirement gathering**: Participate in the software engineering process to gather high-quality specifications[2]. 4**Conduct Formal Technical Reviews**: An FTR is traditionally used to evaluate the quality and design of the prototype[2]. 5**Formulate a Multi-Testing Strategy**: The multi-testing strategy employs different types of testing so that the software product can be tested well from all angles to ensure better quality[2].

8. Discuss Statement Coverage, Conditional Coverage with its formula and applications.

Aspect	Statement Coverage	Conditional Coverage
Definition	Statement Coverage is a white box testing technique that ensures that all the statements of the source code are executed at least once.	Conditional Coverage, also known as Predicate Coverage, ensures that each of the boolean expressions must be evaluated to true and false at least once.
Formula	`Statement Coverage = (Number of executed statements / Total number of statements in source code) * 100.`	`Condition Coverage = (Number of executed operands / Total Number of Operands) * 100.`
Applications	Statement Coverage is used to design test cases where it will find out the total number of executed statements out of the total statements present in the code.	Conditional Coverage is used in software testing to assess the thoroughness of Test Cases.

9. Explain CMM models with applications.

Aspect	Description
Capability Maturity Model (CMM)	The Capability Maturity Model (CMM) is a development model created to improve existing software development processes. It was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987. The model is based on the process maturity framework and describes a strategy for software process improvement that should be followed by moving through 5 different levels. Each level of maturity shows a process capability level.
Applications of CMM	CMM is used to analyze the approach and techniques followed by any organization to develop software products. It also provides guidelines to further enhance the maturity of the process used to develop those software products. It is used in software development and maintenance processes that undergo planning, engineering, and management.
Examples of CMM	
Product Design	The CMM is commonly used for product design.
Mold Equipment	It is used in mold equipment.
Gear and Blade Measurements	CMM is used in gear and blade measurements.
Machinery Manufacturing	It is used in machinery manufacturing.
Tooling Fixtures	CMM is used in tooling fixtures.
Steam Mold Parts	It is used in steam mold parts.
Electronic and Electrical Equipment	CMM is used in electronic and electrical equipment.

10. Explain Control flow testing and data flow testing.

Aspect	Control Flow Testing	Data Flow Testing
Definition	Control Flow Testing is a white box testing technique that uses a program's control flow as a model. It determines and observes the execution paths of a program in a structured way. The goal of control flow testing is to verify that all possible execution paths of a program behave as expected.	Data Flow Testing is a white box testing technique that examines the flow of data in a program. It focuses on the points where variables are defined and used and aims to identify and eliminate potential anomalies that could disrupt the flow of data, leading to program malfunctions or erroneous outputs.
Applications	Control Flow Testing is used to find the following issues: - To find a variable that is used but never defined. - To find a variable that is defined but never used. - To find a variable that is defined multiple times before it is used. - Deallocating a variable before it is used.	Data Flow Testing is used to analyze the flow of data in the program. It is the process of collecting information about how the variables flow the data in the program. It tries to obtain particular information of each particular point in the process.

11. Explain review, inspections and walkthrough.

Aspect	Description
Review	In software testing, a review is a process of examining a software product or component by a group of individuals to detect defects, errors, or vulnerabilities[1][2]. The primary goal of reviews is to identify and fix issues in the early stages of the software development life cycle to save time and cost[1][2]. Reviews can be formal or informal, and they can be carried out at any stage of the software development process[1][2].
Inspections	Software inspection refers to a peer review of software to identify bugs or defects at the early stages of SDLC[3][4][5]. It is a formal review that ensures the documentation produced during a given stage is consistent with previous stages and conforms to preestablished rules and standards[4][5]. Inspections involve people examining the source representation with the aim of discovering anomalies and defects[6]. An inspection does not require execution of a system so may be used before the implementation process[6].

Aspect	Description
Walkthrough	A software walkthrough is a type of peer review where a designer or programmer guides members of the development team and other stakeholders through a software product[7][8][9]. During this process, participants ask questions and comment on potential errors, deviations from development standards, and other issues[7][8][9]. Walkthroughs can be pre-planned or organized based on the needs[9].

12. Discuss check points with examples.

testing in checkpoints refers to the practice of conducting testing activities at specific checkpoints or milestones throughout the development process[1]. These checkpoints are typically defined based on project phases, such as the completion of specific features or modules[1]. Testing in checkpoints helps ensure that the software is functioning correctly and meeting the desired requirements at each stage of development[1]. It involves executing various tests, such as unit testing, integration testing, and system testing, to identify and address any issues or bugs before progressing to the next checkpoint[1].
Here are a few common types of checkpoints[1]:

- **Milestone Checkpoints**: These checkpoints are typically based on project milestones or key deliverables. They mark significant progress points in the development process, such as the completion of a major feature, module, or phase.
- **Feature Checkpoints**: These checkpoints focus on specific features or functionalities within the software. They ensure that each feature is developed, tested, and integrated properly before moving on to the next checkpoint.
- **Integration Checkpoints**: Integration checkpoints verify the successful integration of different components or modules of the software. They ensure that the individual parts of the system work together as intended and are properly integrated into a cohesive whole.
- **Regression Checkpoints**: Regression checkpoints involve running tests to check for any unintended side effects or regression issues introduced by recent changes or updates. They ensure that the software continues to function correctly after modifications and updates have been made.
- **User Acceptance Checkpoints**: User acceptance checkpoints involve testing the software from the end-user's perspective. It allows users or stakeholders to evaluate the software against their requirements and provide feedback before finalizing the development.
- **Performance Checkpoints**: Performance checkpoints focus on evaluating the performance and efficiency of the software. This may involve stress testing, load testing, or performance profiling to ensure the software can handle expected workloads and meet performance expectations.

13. Discuss equivalence partitions with examples.

Aspect	Description
Equivalence Partitioning	Equivalence Partitioning, also known as Equivalence Class Partitioning (ECP), is a black-box testing technique that divides the input domain of a software unit into partitions of equivalent data from which test cases can be derived[1]. This technique is particularly useful when dealing with many input values[2].
Applications of Equivalence Partitioning	Equivalence Partitioning is used to design test cases where it will find out the total number of executed statements out of the total statements present in the code[1]. It is used to check the quality of the code, determine the flow of different paths of the program, and check whether the source code expected to perform is valid or not[1].
Examples of Equivalence Partitioning	Here are some examples of Equivalence Partitioning: 1. **College Admission Process**: Consider a college that gives admissions to students based upon their percentage. The percentage field will accept percentage only between 50 to 90%. More and even less than that will not be accepted, and the application will redirect the user to an error page[1]. 2. **Online Shopping Site**: In an online shopping site, each product has a specific product ID and product name. We can search for a product either by using the name of the product or by product ID. If the product ID entered by the user is invalid then the application will redirect the customer or user to an error page[1]. 3. **Software Application**: Consider a software application that accepts only a particular number of digits, not even greater or less than that particular number. Consider an OTP number that contains only a 6 digit number, greater and even less than six digits will not be accepted, and the application will redirect the customer or user to an error page

14. Explain ISO 9000 models with various versions.

Aspect	Description
ISO 9000	The ISO 9000 family is a set of quality management systems (QMS) standards by the International Organization for Standardization (ISO) that help organizations ensure they meet customer and other stakeholder needs within statutory and regulatory requirements related to a product or service[1]. It was first published in 1987[1]. The ISO 9000 family contains these standards[23]: - **ISO 9001:2015**: Quality Management Systems - Requirements - **ISO 9000:2015**: Quality Management Systems - Fundamentals and Vocabulary (definitions) - **ISO 9004:2018**: Quality Management - Quality of an Organization - Guidance to Achieve Sustained Success (continuous improvement)
Applications of ISO 9000	ISO 9000 is used to analyze the approach and techniques followed by any organization to develop software products[4]. It also provides guidelines to further enhance the maturity of the process used to develop those software products[4]. It is used in software development and maintenance processes that undergo planning, engineering, and management[4].

15. Explain Static Testing.

Aspect	Description
Static Testing	Static Testing is a software testing method that checks for defects in a software application without actually executing the code[1234]. It is performed to ensure that the software is free of defects[4]. In order to discover faults in the code before it is run, static testing is used[4].
Applications of Static Testing	Static Testing is performed at the early stage of development to identify the issues in the project documents in multiple ways, namely reviews, walkthroughs, and inspections[1]. It is process to detect and remove errors and defects in the different supporting documents like software requirements specifications[1]. People examine the documents and sorted out errors, redundancies and ambiguities[1].

16. Explain Junit with examples.

Aspect	Description
JUnit	JUnit is an open-source testing framework for Java programmers[12345]. It allows Java developers to write and execute automated tests[12345]. JUnit provides a set of annotations and assertions that make it easier to write and run tests[2]. The current version is JUnit 4[1].
Applications of JUnit	JUnit is used to perform unit testing in Java[1]. The unit test case is a code which ensures that the program logic works as expected[1]. The org.junit package contains many interfaces and classes for JUnit testing such as Assert, Test, Before, After etc[1].
Examples of JUnit	Here are some examples of JUnit applications: - **Simple JUnit example in Eclipse IDE**: Let's write the logic to find the maximum number for an array[1]. - **JUnit test case example in Java**: We first create the Java code which we want to test, and after that, we will write the test class or test case for our Java code[6].

17. Discuss Junit Annotations with its applications.

Aspect	Description
JUnit Annotations	JUnit Annotations is a special form of syntactic meta-data that can be added to Java source code for better code readability and structure[1]. Variables, parameters, packages, methods, and classes can be annotated[1]. Annotations were introduced in Junit4, which makes Java code more readable and simple[1]. This is the big difference between Junit3 and Junit4 that Junit4 is annotation based[1].

Aspect	Description
Annotation	Description
@Test	This annotation indicates that the public void method to which it is attached can be executed as a test case1.
@Before	This annotation is used if you want to execute some statement such as preconditions before each test case1.
@BeforeClass	This annotation is used if you want to execute some statements before all the test cases1.
@After	This annotation can be used if you want to execute some statements after each Test Case1.
@AfterClass	This annotation can be used if you want to execute some statements after all test cases1.
@Ignores	This annotation can be used if you want to ignore some statements during test execution1.
@Test(timeout=500)	This annotation can be used if you want to set some timeout during test execution1.
@Test(expected=IllegalArgumentException.class)	This annotation can be used if you want to handle some exception during test execution1.

18. Discuss Junit with simple programs to verify unit and integration testing.

JUnit is a popular testing framework for Java. It provides annotations to identify test methods and contains many assertions for checking the expected results

Here's a simple example of a unit test in JUnit:

Java

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {
   @Test
   public void testAdd() {
      String str = "JUnit is working fine";
      assertEquals("JUnit is working fine", str);
   }
}
```

For integration testing, JUnit can also be used. Integration testing is where individual units are tested together to check whether all the units interact with each other as expected[1]. Here's a simple example:

Java

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class IntegrationTest {
   @Test
   public void testIntegration() {
      // Setup the classes to be integrated
      ClassA classA = new ClassA();
      ClassB classB = new ClassB();

      // Perform the integration test
      boolean result = classA.methodA(classB);
      assertEquals(true, result);
   }
}
```

19. Discuss Selenium.

Selenium is a free, open-source automated testing framework used to validate web applications across different browsers and platforms[12]. It was originally developed by Jason Huggins in 2004 as an internal tool at Thought Works[3].

Here are the key components of Selenium:

1. **Selenium WebDriver**: If you want to create robust, browser-based regression automation suites and tests, scale and distribute scripts across many environments, then you want to use Selenium WebDriver[1]. It provides a collection of language-specific bindings to drive a browser[1].
2. **Selenium IDE**: If you want to create quick bug reproduction scripts, create scripts to aid in automation-aided exploratory testing, then you want to use Selenium IDE[1]. It is a Chrome, Firefox, and Edge add-on that will do simple record-and-playback of interactions with the browser[1].
3. **Selenium Grid**: If you want to scale by distributing and running tests on several machines and manage multiple environments from a central point, making it easy to run the tests against a vast combination of browsers/OS, then you want to use Selenium Grid[1].

Selenium supports a variety of programming languages through the use of drivers specific to each language. Languages supported by Selenium include C#, Java, Perl, PHP, Python, and Ruby[3]. Selenium test scripts can be coded in any of the supported programming languages and can be run directly in most modern web browsers[3]. Selenium can be used to automate functional tests and can be integrated with automation test tools such as Maven, Jenkins, & Docker to achieve continuous testing. It can also be integrated with tools such as TestNG, & JUnit for managing test cases and generating reports[3].

In summary, Selenium is a powerful tool for controlling a web browser through the program. It's used for automating web applications for testing purposes, but it's not limited to just that. Boring web-based administration tasks can (and should) also be automated[1].

20. How software bugs affects the organizations like Meta or Google or banking systems.

Software bugs can have significant impacts on organizations like Meta, Google, or banking systems. Here are some potential effects:

1. **Security Breaches**: Software bugs can lead to security vulnerabilities, which can be exploited by malicious actors. For example, a data breach at T-Mobile affected 50 million customers due to a software bug[1]. Such breaches can lead to the loss of sensitive customer data, including names, addresses, and even social security numbers[1].
2. **Financial Losses**: The direct financial impact of software bugs can be substantial. This can include the costs of fixing the bugs, potential regulatory fines, and compensation for affected customers. In the case of the T-Mobile data breach, many customers affected by the data breach decided to take legal action[1].
3. **Reputation Damage**: Software bugs can damage an organization's reputation. Customers may lose trust in the company's ability to protect their data, which can lead to a loss of business. For example, after the T-Mobile data breach, the company's security protocols were criticized, and the CEO had to apologize and promise to improve defenses[1].
4. **Operational Disruptions**: Software bugs can cause disruptions to a company's operations. For example, a bug in a new feature of Slack led to backlash from users and required the company to make adjustments[1].
5. **Increased Costs**: Organizations like Meta invest heavily in finding and fixing bugs as part of their "defense-in-depth" security strategy[2]. This includes automated tooling, security reviews, red teaming, and bug bounty programs[2]. These activities represent a significant ongoing cost.

In summary, while software bugs are often unavoidable, their impact can be mitigated through robust testing, security practices, and response strategies. It's important for organizations to invest in these areas to protect their customers, operations, and reputation.

21. Explain Du, Dc, DD graph, path graph and cyclomatic complexity with applications.

Concept	Explanation
Du Graph (Definition-Use Graph)	This is a method used in data flow testing. The Du graph focuses on "All Definition-Use Paths". Using this technique, every possible path from a variable's definition to every usage point is tested. It helps in minimizing computation time and provides an easy way to detect software failures and bugs in a code during execution.
Dc Graph	I couldn't find specific information on a "Dc Graph" in the context of software testing. It might be a typo or a less common term. If you could provide more context, I might be able to give a more accurate answer.
DD Graph (Decision-to-	A DD graph is used to find independent paths for testing. Every statement in the program has been executed at least once. DD-path testing is a well-

Concept	Explanation
Decision Graph)	known code-based testing method.
Path Graph	Path Testing is a method used to design test cases. In this method, the control flow graph of a program is designed to find a set of linearly independent paths of execution. Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.
Cyclomatic Complexity	Cyclomatic Complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths in the source code of a software program. It is computed using the Control Flow Graph of the program. It can be used as a quality metric, given the relative complexity of various designs.

Applications of these concepts:

Concept	Application
Du Graph	It is used in data flow testing to find the test paths of a program according to the locations of definitions and uses of variables in the program.
Dc Graph	If you could provide more context, I might be able to give a more accurate answer.
DD Graph	It is used to find independent paths for testing. It ensures that every statement in the program has been executed at least once.
Path Graph	It is used in path testing to design test cases. It helps in achieving complete branch coverage without covering all possible paths of the control flow graph.
Cyclomatic Complexity	It is used to measure the complexity of a program. It helps in determining the number of test cases to achieve branch coverage. It is also used in path testing to find every possible executable path.

22. Explain Boundary value analysis and robust case analysis of the triangle program/Next date program.

Triangle Program:

Boundary Value Analysis (BVA):

- Inputs: Three side lengths (positive numbers).
- Boundaries:
  - Valid triangle: 0 < side1 < side2 + side3 (and similar permutations).
  - Invalid triangle: Any violation of the valid triangle inequality.
- Test Cases:
  - Valid: (1, 2, 3), (4, 5, 6), (0.1, 0.2, 0.3) -- Test basic valid triangles.
  - Invalid: (1, 10, 1), (1, 1, 2), (999, 1, 1) -- Test invalid triangles due to inequality violations.
  - Zero value: (0, 1, 2), (1, 0, 2), (1, 2, 0) -- Test cases with zero values.
  - Equal values: (3, 3, 3), (1.5, 1.5, 1.5) -- Test equilateral triangles.
  - Large values: (1000, 500, 300), (99999, 99999, 99998) -- Test for potential integer overflow or precision issues.

Robust Case Analysis:

- Scenarios:
  - Degenerate triangles: (0, 0, 0), (1, 2, 1000) -- Test if program handles unexpected or degenerate cases gracefully.
  - Floating-point precision: Use values like 1.000000000000001 to test robustness against floating-point limitations.
  - Non-numeric inputs: (1, "abc", 3), ("hello", 2, 5) -- Test error handling for non-numeric inputs.

Next Date Program:

Boundary Value Analysis (BVA):

- Inputs: Day (1-31), month (1-12), year (some range, e.g., 1900-2100).
- Boundaries:
  - Valid dates: All valid combinations within ranges.
  - Invalid dates: Non-existent dates (February 30th), out-of-range values.
  - Leap years: February 29th in leap years.
  - Month end: Dates like January 31st, December 31st.

    - Year change: December 31st to January 1st across different years.
- Test Cases:
    - Valid: (15, 5, 2023), (31, 12, 2022), (29, 2, 2024) -- Test basic valid dates, including leap year.
    - Invalid: (32, 5, 2023), (0, 8, 2024), (15, 13, 2023) -- Test invalid dates due to out-of-range values.
    - Month end: (31, 1, 2024), (30, 9, 2023) -- Test handling of month end.
    - Year change: (31, 12, 2023), (1, 1, 2024) -- Test handling of year change.

Robust Case Analysis:

- Scenarios:
    - Edge cases: December 31st in non-leap years, February 28th in leap years.
    - Large or small years: Test for potential integer overflow or underflow issues.
    - Historical or future dates: Test if program handles dates outside the expected range.
    - Invalid input formats: Test error handling for non-numeric or out-of-format inputs.

23. Explain equivalence class analysis of triangle program or next date program.

the equivalence class analysis for both the Triangle Program and the Next Date Program:

Triangle Program:
Equivalence class testing is a black-box testing technique used to create test cases. It divides the input data of a software unit into partitions of equivalent data from which test cases can be derived[1]. In principle, test cases are designed to cover each partition at least once[1].
For the Triangle Program, the input domain can be divided into the following equivalence classes[2]:

1. Equilateral triangles (all sides equal)
2. Isosceles triangles (two sides equal)
3. Scalene triangles (no sides equal)
4. Not a triangle (the sum of the lengths of any two sides is less than or equal to the length of the third side)

Test cases would then be derived to cover each of these classes[2].

Next Date Program:
For the Next Date Program, the input domain can be divided into the following equivalence classes[3]:

1. Day: 1 to 31

2. Month: 1 to 12
3. Year: 1812 to 2012

The output domain can be divided into the following equivalence classes[3]:

1. Valid next date
2. Invalid input date

Test cases would then be derived to cover each of these classes[3]. For example, testing the transition from the end of one month to the beginning of the next, or the transition from February 28 to February 29 in a leap year[3].

These test cases help ensure that the software handles all possible inputs correctly and can provide valuable information about its robustness and reliability.

24. Find the Cyclomatic Complexity for the C program segment:-

```
while (first <= last)
{
if (array [middle] < search)
first = middle +1;
else if (array [middle] == search)
found = True;
else last = middle – 1;
middle = (first + last)/2;
}
if (first < last) not Present = True;
```

Cyclomatic Complexity (CC) is a software metric used to measure the complexity of a program. It is computed using the Control Flow Graph (CFG) of the program and is calculated as:

$CC=E-N+2P$

where:

- E is the number of edges in the CFG.
- N is the number of nodes in the CFG.
- P is the number of connected components.

For the given code segment, let's calculate the Cyclomatic Complexity:

1. The `while` loop is a decision point.
2. The `if` statement inside the `while` loop is another decision point with three outcomes.
3. The `if` statement after the `while` loop is another decision point.

So, there are 3 decision points. According to McCabe's Cyclomatic Complexity, the complexity is calculated as:

CC=Number of Decision Points+1

So, the Cyclomatic Complexity of the given code segment is:

CC=3+1=4

Therefore, the Cyclomatic Complexity of the given code segment is **4**. This means there are 4 linearly independent paths through the code.

25. Find the Cyclomatic Complexity for the following Code:

```
while (m<n)
if (x>y ) and (a<b) then
a=a+1
y=y-1
end if
m=m+1 end while
```

Cyclomatic Complexity (CC) is a software metric used to measure the complexity of a program. It is computed using the Control Flow Graph (CFG) of the program and is calculated as:

CC=E−N+2P

where:

- E is the number of edges in the CFG.
- N is the number of nodes in the CFG.
- P is the number of connected components.

For the given code segment, let's calculate the Cyclomatic Complexity:

1. The `while` loop is a decision point.
2. The `if` statement inside the `while` loop is another decision point.

So, there are 2 decision points. According to McCabe's Cyclomatic Complexity, the complexity is calculated as:

CC=Number of Decision Points+1

So, the Cyclomatic Complexity of the given code segment is:

CC=2+1=3

Therefore, the Cyclomatic Complexity of the given code segment is **3**. This means there are 3 linearly independent paths through the code.

26. Consider the following method :

Find the test sets for Full Statement Coverage and Full Branch Coverage.

```
int f(int m, int n, boolean x, boolean y)
{
int res=0;
if(m<0) {res=n-m;}
else if(x || y) {
res= -1;
if(n==m) {res =1;}
}
else {res=n;}
return res;
} /*end of f */
```

the test sets for Full Statement Coverage and Full Branch Coverage for the given method.

Full Statement Coverage:

This requires that every line of code is executed at least once. Here are the test cases that will ensure full statement coverage:

1. `f(-1, 1, true, true)`: This will cover the `if(m<0)` branch.
2. `f(1, 1, true, true)`: This will cover the `else if(x || y)` branch and the `if(n==m)` branch inside it.
3. `f(1, 2, false, false)`: This will cover the `else` branch.

Full Branch Coverage:

This requires that every branch (both the true and false conditions) is covered. Here are the test cases that will ensure full branch coverage:

1. `f(-1, 1, true, true)`: This will cover the `if(m<0)` branch (true condition).
2. `f(1, 1, true, true)`: This will cover the `else if(x || y)` branch (true condition) and the `if(n==m)` branch inside it (true condition).
3. `f(1, 2, true, true)`: This will cover the `if(n==m)` branch inside the `else if(x || y)` branch (false condition).
4. `f(1, 2, false, false)`: This will cover the `else if(x || y)` branch (false condition) and the `else` branch.

27. Write a program/main function in C/Java to enter a date, month and year and find out if it's a valid date or not. Write test cases to for Full Code Coverage of the Code for the program you built.

C program that checks if a given date is valid or not:

```
#include <stdio.h>

int isValidDate(int day, int month, int year) {
    if(year < 1800 || year > 9999)
        return 0;
    if(month < 1 || month > 12)
        return 0;
    if(day < 1 || day > 31)
        return 0;

    // Handle February month with leap year
    if(month == 2) {
        if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)) {
            return day <= 29;
        } else {
            return day <= 28;
        }
    }

    // Months of April, June, September and November have maximum of 30
days
    if(month == 4 || month == 6 || month == 9 || month == 11)
        return day <= 30;

    return 1;
}

int main() {
    printf("%d\n", isValidDate(29, 2, 2020)); // Output: 1
    printf("%d\n", isValidDate(29, 2, 2021)); // Output: 0
    return 0;
}
```

Now, let's write some test cases for full code coverage:

```
int main() {
    assert(isValidDate(1, 1, 2000) == 1); // A valid date
    assert(isValidDate(29, 2, 2020) == 1); // A valid leap year date
    assert(isValidDate(29, 2, 2021) == 0); // An invalid leap year date
    assert(isValidDate(31, 4, 2020) == 0); // An invalid date (April has 30 days)
    assert(isValidDate(32, 1, 2020) == 0); // An invalid date (No month has 32
days)
    assert(isValidDate(1, 13, 2020) == 0); // An invalid date (There are only 12
months)
    assert(isValidDate(1, 1, 10000) == 0); // An invalid date (Year out of range)
    return 0;
}
```

28. Write a test case in Junit and Test it using Junit framework and libraries. You can use the following tutorials regarding Junit from this links:-
    1. https://www.javatpoint.com/junit-tutorial
    2. https://www.guru99.com/create-junit-test-suite.html
    3. https://www.tutorialspoint.com/junit/junit_suite_test.htm

a simple JUnit test case. Let's say we have a simple Java class called `Calculator` that we want to test:

Java

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

a JUnit test case for the `add` method in the `Calculator` class like this:

Java

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(5, 3);
        assertEquals(8, result);
    }
}
```