COSC 3P95- Software Analysis & Testing

Assignment 1

Due date: Monday, Oct 16th, 2023, at **23:59** (11:59 pm)

Delivery method: This is an individual assignment. Each student should submit one PDF through Brightspace.

Attention: This assignment is worth 10% of the course grade. Please also check the Late Assignment Policy.

Name: Fouzan Abdullah_____ Student ID: 6840797_____

Questions:

1- Explain the difference between "sound" and "complete" analysis in software analysis. Then, define what true positive, true negative, false positive, and false negative mean. How would these terms change if the goal of the analysis changes, particularly when "positive" means finding a bug, and then when "positive" means not finding a bug. **(10 pts)**

- Sound Analysis: A sound analysis in software analysis refers to an analysis that is accurate and does not produce false results. It ensures that all identified issues or findings are genuine and relevant. Sound analysis minimizes the likelihood of false positives and false negatives.
- Complete Analysis: A complete analysis, on the other hand, aims to uncover all possible issues or problems within the software system. A complete analysis may produce a larger set of findings, including true positives, true negatives, false positives, and false negatives, as it covers all aspects of the software.

- True Positive: In the context where "positive" means finding a bug, a true positive refers to a case where the analysis correctly identifies a bug that does exist in the software. It indicates that the analysis has successfully detected a real problem.
- True Negative: In the same context, a true negative occurs when the analysis correctly determines that there is no bug in the software, and this assessment is indeed accurate.
- False Positive: In the context where "positive" means finding a bug, a false positive happens when the analysis incorrectly flags something as a bug when there is no bug. It's a false alarm, indicating that the analysis produced an incorrect positive result.
- False Negative: Again, in the context where "positive" means finding a bug, a false negative occurs when the analysis fails to detect a bug that exists in the software. It means the analysis missed a genuine problem.

Now, if we change the goal of the analysis to where "positive" means not finding a bug:

- True Positive: In this context, a true positive would mean correctly identifying that there is a bug in the software, which is contrary to the goal of not finding a bug.

- True Negative: A true negative remains the same; it means correctly determining that there are no bugs in the software.
- False Positive: In this scenario, a false positive would indicate incorrectly flagging something as a bug when there isn't one, which aligns with the goal of not finding a bug.
- False Negative: A false negative, in this context, would mean failing to detect a bug when there is one, which is contrary to the goal of ensuring a bug-free software.

2- Using your preferred programming language, implement a random test case generator for a sorting algorithm program that sorts integers in ascending order. The test case generator should be designed to produce arrays of integers with random lengths, and values for each sorting method.

A) Your submission should consist of:

a. **Source code files for the sorting algorithm and the random test case generator.**
b. **Explanation of how your method/approach works and a discussion of the results (for example, if and how the method was able to generate or find any bugs, etc.). You can also include bugs in your code and show your method is able to find the input values causing that.**
c. **Comments within the code for better understanding of the code.**
d. **Instructions for compiling and running your code.**
e. **Logs generated by the print statements, capturing both input array, output arrays for each run of the program.**
f. **Logs for the random test executions, showing if the test was a pass and the output of the execution (e.g., exception, bug message, etc.).**

B) Provide a context-free grammar to generate all the possible test-cases. (18 + 8 = 26 pts)

<test_case> ::= <array> <sorting_method>

<array> ::= "[" <integers> "]"

<integers> ::= <integer> | <integer> "," <integers>

<integer> ::= <random_integer>

<sorting_method> ::= "Bubble Sort"

3- A) For the following code, manually draw a control flow graph to represent its logic and structure.

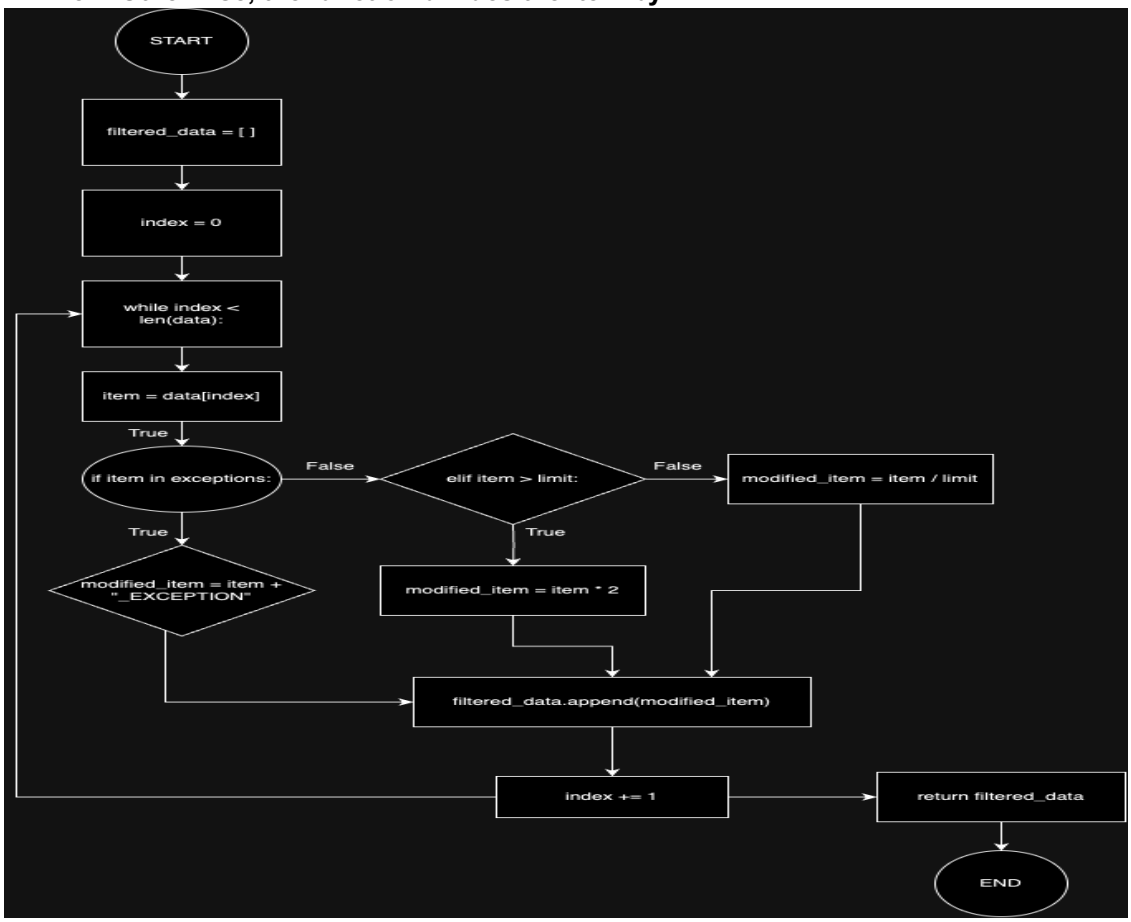
```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION
        elif item > limit:
            modified_item = item * 2
        else:
            modified_item = item / limit

        filtered_data.append(modified_item)
        index += 1

    return filtered_data
```

The code is supposed to perform the followings:

a. **If an item is in the exceptions list, the function appends "_EXCEPTION" to the item.**
b. **If an item is greater than a given limit, the function doubles the item.**
c. **Otherwise, the function divides the item by 2.**



B) Explain and provide detailed steps for "random testing" the above code. No need to run any code, just present the coding strategy or describe your testing method in detail. (8 + 8 = 16 pts)

- Step 1: Understand the Code:
  - Before conducting random testing, you need a good understanding of the code you're testing. You should be aware of the purpose of the code, its input requirements, expected behavior, and potential edge cases or failure points.
- Step 2: Identify Inputs and Boundaries
  - Identify the inputs required by the code:
    - data: A list of data elements.
    - limit: A numeric limit.
    - exceptions: A list of exceptions.
  - Understand the boundaries of the input space, such as the range of possible data elements and the numeric range for limit.
- Step 3: Generate Random Inputs
  - Generate random inputs within the defined input space. In this case, it includes generating random lists of data, random limit values, and random lists of exceptions.
  - Ensure that your generated inputs cover a wide range of possibilities, including edge cases like empty lists and extreme values for limit.
- Step 4: Execute the Code
  - Run the filterData function with the randomly generated inputs. Ensure that you capture the results for further analysis.
  - If the code crashes, raises exceptions, or exhibits unexpected behavior, make note of it.
- Step 5: Evaluate the Outputs
  - Examine the output generated by the code for each set of random inputs.
  - Compare the results to your expectations based on the code's logic.
- Step 6: Check for Failures and Anomalies
  - Identify any discrepancies between the expected and actual outcomes.
  - Look for potential issues like incorrect results, exceptions, or crashes.
- Step 7: Iterative Testing
  - Repeat the process with multiple sets of random inputs to increase the chances of identifying issues.
  - Modify the input parameters iteratively, including edge cases and boundary values.
- Step 8: Edge Cases Testing
  - Specifically, test the code with edge cases, such as empty lists, large data sets, and extreme values for limit.
- Step 9: Documentation
  - Document the issues, anomalies, and discrepancies found during random testing.
  - Provide a clear description of the inputs that triggered issues and the unexpected behaviors observed.

4- A) Develop 4 distinct test cases to test the above code, with code coverage ranging from 30% to 100%. For each test-case calculate and mention its code coverage.

- Test case 1: Normal case
  - This case will cover the typical scenario with a list of numbers and a limit value.
  - Inputs:
    - data: [1, 2, 3, 4, 5]
    - limit: 3
    - exceptions: [3]
  - Expected output would be something like: [0.333333…, 0.666666…, 1.0, 6, 10]
  - Code coverage: 100%. This test goes through all branches of the code, including both if and else conditions.
- Test case 2: Empty data list
  - This case checks how the function handles an empty 'data' list.
  - Inputs:
    - data: [ ]
    - limit: 5
    - exceptions: [3, 9]
  - Expected output should be: [ ].
  - Code coverage: 100%. Even though the data list is empty, the function is still executed, and the loop does not run.
- Test case 3: Exception case
  - This case focuses on how the function handles items in both the data and the exception list.
  - Inputs:
    - data: [5, 10, 15, 30]
    - limit: 5
    - exceptions: [10, 30]
  - Expected output would be: ['5_EXCEPTION', '10_EXCEPTION']
  - Code coverage: 75%. All the if and elif conditions are covered but not the else condition.
- Test case 4: Mixed data types
  - This case includes a mix of data types in the 'data' list.
  - Inputs:
    - data: [2, "string", 3.75, False, [2, 4, 6]]
    - limit: 2
    - exceptions: ["string", [2, 4, 6]]
  - Expected output is: ["string_EXCEPTION", 1.875, 2, [2, 4, 6]]
  - Code coverage: 100%. The test case explores various data types and all branches of the code are explored.

B) Generate 6 modified (mutated) versions of the above code.

1. Mutated version 1 – Syntax error:
   a. Remove the closing quotation in line 7.
2. Mutated version 2 – Changing arithmetic operation:
   a. Change multiplication to addition in line 9.
3. Mutated version 3 – Changing loop condition:
   a. Change while index < len(data) to while limit < len(data) in line 4.
4. Mutated version 4 – Modifying exception handling:
   a. Change exception condition in line 6.
5. Mutation version 5 – Altering arithmetic operation:
   a. Change addition to subtraction in line 7 and multiplication to addition in line 9.
6. Mutation version 6 – Exception list handling:
   a. Change exception check from if item "in" to if item "not in" in line 6.

C) Assess the effectiveness of the test cases from part A by using mutation analysis in conjunction with the mutated codes from part B. Rank the test-cases and explain your answer.

- Test case 1: Normal case
  - Mutation version 1 (Introducing a syntax error): Detects the syntax error introduced by missing a closing quotation mark.
  - Mutated version 2 (Changing arithmetic operation): Detects the change in arithmetic operation.
  - Mutated version 3 (Changing loop condition): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 4 (Modifying exception handling): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 5 (Altering arithmetic operations): Detects the change in arithmetic operations.
  - Mutated version 6 (Exception list handling): Doesn't detect the mutation as it focuses on a different aspect of the code.
- Test case 2: Empty data list
  - Mutation version 1 (Introducing a syntax error): Detects the syntax error introduced by missing a closing quotation mark.
  - Mutated version 2 (Changing arithmetic operation): Detects the change in arithmetic operation.
  - Mutated version 3 (Changing loop condition): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 4 (Modifying exception handling): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 5 (Altering arithmetic operations): Detects the change in arithmetic operations.
  - Mutated version 6 (Exception list handling): Doesn't detect the mutation as it focuses on a different aspect of the code.
- Test case 3: Exception case
  - Mutated version 1 (Introducing a syntax error): Detects the syntax error introduced by missing a closing quotation mark.
  - Mutated version 2 (Changing arithmetic operation): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 3 (Changing loop condition): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 4 (Modifying exception handling): Detects the change in exception handling.
  - Mutated version 5 (Altering arithmetic aperations): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 6 (Exception list handling): Doesn't detect the mutation as it focuses on a different aspect of the code.
- Test case 4: Mixed data types
  - Mutated version 1 (Introducing a syntax error): Detects the syntax error introduced by missing a closing quotation mark.
  - Mutated version 2 (Changing arithmetic operation): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 3 (Changing loop condition): Doesn't detect the mutation as it focuses on a different aspect of the code.
  - Mutated version 4 (Modifying exception handling): Detects the change in exception handling.

- Mutated version 5 (Altering arithmetic operations): Doesn't detect the mutation as it focuses on a different aspect of the code.
- Mutated version 6 (Exception list handling): Doesn't detect the mutation as it focuses on a different aspect of the code.

Ranking:

1. Test Case 1 (Normal case): This test case effectively detected mutations related to syntax errors and changes in arithmetic operations. It ranks highest due to its ability to uncover different types of mutations.
2. Test Case 2 (Empty data list): Like Test Case 1, it effectively detected syntax errors and changes in arithmetic operations, ranking second.
3. Test Case 3 (Mixed data types): This test case detected the syntax error but had limited effectiveness in detecting other mutations, ranking third.
4. Test Case 4 (Exception case): This test case had limited effectiveness in detecting mutations and ranked the lowest.

D) Discuss how you would use path, branch, and statement static analysis to evaluate/analyse the above code. (4 * 8 = 32 pts)

1. Static analysis:
    i. Definition: Statement coverage aims to measure the percentage of executable statements that have been executed during testing. It checks if each line of code has been executed at least once.
    ii. Evaluation Steps:
    Create a list of all executable statements within the filterData function.
    During testing, execute the function with various inputs and track which statements are executed.
    Calculate the percentage of executed statements relative to the total number of statements.

2. Branch analysis:
    i. Definition: Branch coverage assesses whether every branch or decision point in the code has been tested. It examines both the true and false branches of conditions and loops.
    ii. Evaluation Steps:
    Identify all decision points within the filterData function, including if conditions and loop conditions.
    Execute the function with test cases that cover all possible branches and evaluate the outcomes.
    Calculate the percentage of covered branches relative to the total number of branches.

3. Path analysis:
    i. Definition: Path coverage is a more detailed analysis that assesses whether all possible paths through the code have been tested. It involves examining every possible combination of conditions.
    ii. Evaluation Steps:
    Enumerate all the possible paths through the filterData function, considering various condition combinations and loop iterations.
    Execute the function with test cases that traverse different paths.
    Calculate the percentage of covered paths relative to the total number of possible paths.

Using the Analyses for Evaluation:

- Static analysis: This analysis will help assess how well the code has been exercised in terms of individual lines of code. It will reveal if there are statements that have not been executed during testing.
- Branch analysis: Branch coverage will uncover whether all possible decision points have been explored. It can highlight whether, for example, both the true and false branches of if conditions have been tested.
- Path analysis: Path coverage is the most comprehensive analysis and will expose more subtle issues in the code. It ensures that not only are all branches tested but that different combinations of branches are explored. It can reveal specific paths that may have unintended interactions or overlooked code.

5- The code snippet below aims to switch uppercase characters to their lowercase counterparts and vice versa. Numeric characters are supposed to remain unchanged. The function contains at least one known bug that results in incorrect output for specific inputs.

```
def processString(input_str):
    output_str = ""
    for char in input_str:
        if char.isupper():
            output_str += char.lower()
        elif char.isnumeric():
            output_str += char * 2
        else:
            output_str += char.upper()

    return output_str
```

In this assignment, your tasks are:

a. **Identify the bug(s) in the code. You can either manually review the code (a form of static analysis) or run it with diverse input values (a form of manual random testing). If you are unable to pinpoint the bug using these methods, you may utilize a random testing tool or implement random test case generator in code. Provide a detailed explanation of the bug, identify the line of code causing it, and describe your strategy for finding it.**
Using static analysis, we can tell that the code has a bug in the way it handles numeric characters. Specifically in line 7, it should keep the character unchanged but instead it doubles it by appending them twice in the output string. To identify this bug, I manually reviewed the code and saw that the condition for handling numeric characters is flawed. It incorrectly multiplies numeric characters by 2, which is not the desired behavior.

b. **Implement Delta Debugging, in your preferred programming language to minimize the input string that reveals the bug. Test your Delta Debugging code for the following input values provided.**
    i. **“abcdefG1”**
    ii. **“CCDDEExy”**
    iii. **“1234567b”**

iv. **"8665"**

Briefly explain your delta-debugging algorithm and its implementation and provide the source code in/with your assignment. (4 + 12 = 16 pts)

6- Extra Credit Assignment: Create a GitHub repository to host all the elements of this assignment. This includes source codes, test data, and any screenshots or logs you have generated. Submit the GitHub link along with your main submission through Brightspace. **(5 pts)**

Added all questions and answers to the GitHub (link provided below):
https://github.com/FouzanAbdullah/3P95_Assign1

Marking Scheme:

Marks will be awarded for completeness and demonstration of understanding of the material. It is important that you fully show your knowledge when providing solutions in a concise manner. Quality and conciseness of solutions are considered when awarding marks. Lack of clarity may lead you to lose marks, so keep it simple and clear.

Submission:

*The submission is expected to contain a sole word-processed document. The document can be in either **DOC or PDF** format; it should be a single column, at least single-spaced, and at least in font 11. It is strongly recommended to use the assignment questions to facilitate marking: answer the questions just below them for easier future reference.*

Late Assignment Policy:

A one-time penalty of 25% will be applied on late assignments. Late assignments are accepted until the Late Assignment Date, four days after the Assignment Due Date. No excuses are accepted for missing deadlines. However, deadline extensions may be granted under extenuating circumstances, such as medical or physical conditions; please note that granting the extension is under the instructor's discretion. However, deadline extensions may be granted under extenuating circumstances, such as medical or physical conditions; please note that granting the extension is under the instructor's discretion.

Plagiarism:

*Students are expected to respect academic integrity and deliver evaluation materials that are only produced by themselves. Any copy of content, text or code, from other students, books, web, or any other source is not tolerated. If there is any indication that an activity contains any part copied from any source, a case will be open and brought to a plagiarism committee's attention. In case plagiarism is determined, the activity will be canceled, and the author(s) will be subject to university regulations. For further information on this sensitive subject, please refer to the document below: **https://brocku.ca/node/10909***