Basim Ahmad

7022494

COSC 3P95 Assign1

Q1.

**Sound Analysis:** In software analysis, "sound analysis" refers to an analysis that ensures accuracy. To put it another way, a sound analysis is one that yields accurate results every time—neither false positives nor false negatives.

**Complete Analysis:** On the other hand, thorough analysis looks for any and all problems or outcomes within the software being examined. Although it makes sure nothing is overlooked, it may accept some false positives or false negatives.

**True Positive:** A situation in which a relevant issue or defect is appropriately identified as a problem by a study of software.

**True Negatives:** When an analysis correctly determines that a particular situation is not a problem, and it is in fact not a software problem.

**False Positive:** A false positive occurs when the analysis incorrectly flags something as an issue or bug when it is not actually a problem in the software.

**False Negative:** A false negative happens when the analysis fails to detect a genuine issue or bug in the software, and it incorrectly determines that there is no problem when there is one.

**When "Positive" Means Finding a Bug:** If the goal of the analysis is to find bugs, the interpretation of false positives and false negatives would change. In this context:

**False Positive:** It would represent cases where the analysis mistakenly identifies something as a bug when it is not.

**False Negative:** It would represent cases where the analysis fails to identify a bug when there is one in the software.

True positive and true Negative remain the same.

**When "Positive" Means Not Finding a Bug:** If the goal of the analysis is to ensure that no bugs are present, then the interpretation changes further:

**True Positive:** In this case, a true positive would represent correctly identifying that something is not a bug when it's not a bug.

**True Negative:** It would mean correctly determining that something is a bug when it is a bug.

**False Positive:** A false positive would now represent cases where the analysis mistakenly identifies something as a bug when it is not.

**False Negative:** It would mean identifying that something is not a bug when it is a bug.

Q2.



Test description -> "Test"

Input description -> "Input:"

Array size -> "Array size" <integer>

max value -> "Maximum value" <integer>

test number -> "Test number" <integer>

sorting algorithm -> "Sorting algorithm:"

Q3. (a)

A screenshot of a computer

Description automatically generated

(b)

**Step 1:**

Specify the precise goals and intended actions of the code. In this instance, the goals are to evaluate how the code handles various kinds of input data:

Things that are listed as exceptions.

Items in excess of the limit.

Items that neither exceed the limit nor are on the exceptions list.

**Step 2:**

Choose the attributes and range of the input data you want to test. The input area in this code consists of:

the data set's integer-only data array.

the upper bound.

The list of exceptions

**Step 3:**

For every test case in the specified input space, provide random test case input data.

**Step 4:**

Run the programme using the randomly generated inputs for each test scenario.

**Step 5:**

Verify the filterData function's output for each test scenario. Based on the specified objectives, determine whether the output contains the expected transformations.

**Step 6:**

Keep track of each test case's outcomes, including the input information, the anticipated and actual results, and any deviations from the planned course of action.

**Step 7:**

If problems are found, fix the code, and run more random tests to make sure the problems have been fixed. This approach should be repeated until the code responds to a variety of random inputs as predicted.

Q4.

(a)

**Test Case 1: 30% Coverage**

data = [1, 2, 3, 4]

limit = 3

exceptions = [2, 4]

filtered\_result = filterData(data, limit, exceptions)

By including both numbers that are smaller than the limit and numbers that are listed in the exceptions, this test case verifies the fundamental functioning. The "item > limit" condition isn't covered, but it does handle the "if," "elif," and "else" branches.

**Test Case 2: 60% Coverage**

data = [1, 2, 3, 4]

limit = 2

exceptions = [2, 4]

filtered\_result = filterData(data, limit, exceptions)

This test case includes exceptions and looks for numbers more than the maximum. It covers the "item > limit" condition, the "if" and "elif" branches, but not the "else" branch.

**Test Case 3: 80% Coverage**

data = [1, 2, 3, 4]

limit = 5

exceptions = [2, 4]

filtered\_result = filterData(data, limit, exceptions)

Numbers below the limit, the "item > limit" condition, and exceptions are all included in this test case. Except for the "else" branch, it applies to all code branches.

**Test 4: 100% Coverage**

data = [1, 2, 3, 4]

limit = 5

exceptions = [2, 4, 6]

filtered\_result = filterData(data, limit, exceptions)

This test case incorporates each item from the data list, guaranteeing that the entire piece of code is run. It provides 100% coverage by encompassing all code lines and branches.

(b)

**Mutation1:** The string concatenation operation in the "if" branch was altered in this alteration from "+" to "+\_EXCEPTION."

**Mutation 2:** changed the "item > limit" condition to "item >= limit."

**Mutation 3:** changed the "item > limit" condition to "item < limit."

**Mutation 4:** changed the "else" branch from division to subtraction.

**Mutation 5:** changed the "else" branch from division to multiplication.

**Mutation 6:** changed the divisor in the "else" branch to "limit + 1" for division.

(c)

**Test Case 4**: It should be very good at spotting mutations because it has 100% code coverage. It ought to be able to recognise the majority, if not all, of the changes made to the code.

**Test Case 3**: Because this test case includes a variety of scenarios, it should be successful in identifying mutations. However, since the "else" branch is not covered, mutations there can go unnoticed.

**Test Case 2**: It includes the "if" and "elif" branches as well as the "item > limit" condition, and it covers a sizable chunk of the code. It might be just marginally successful in finding mutations, but it won't pick up changes in the "else" branch.

**Test Case 1**: This test case has the lowest sensitivity and coverage in terms of mutation detection. Only mutations in the "if" and "elif" branches may be found using it.

(d)

**Static Analysis Coverage:**

The percentage of statements that have been executed during testing is known as statement coverage. It shows which code statements have been executed and which have not.

How to use:

Run the code through each of the test cases from section A.

Keep track of the statements that are run each time the code is executed.

To display which statements are executed (covered) and which are not, create a coverage report.

**Branch Coverage:**

Branch coverage measures the extent to which a decision (if-else) statement has used each of its potential branches. It makes sure that every route through the code is considered.

How to use:

Run your test cases and monitor which code branches (such as the "if," "elif," and "else" branches) are executed.

To see which branches have been covered and which have not, create a coverage report.

**Path Coverage:**

Path coverage assesses whether each and every route through the code has been taken. Due to the fact that it takes into account all possible branch and loop combinations, it is more thorough than branch coverage.

How to use:

Determine manually all feasible routes through the code, taking into account all potential arrangements of conditional branching and loops.

Run your test cases to see if they adhere to the defined pathways.

**Static Coverage :** By running the test cases, you can find out if any statements are never carried out. You would know which sentences are in the "else" block and are not executed, for example, if the "else" branch is never entered.

**Branch Coverage:** You can check to see if conditional statements test every possible branch. You can, for instance, validate that testing always includes the "if," "elif," and "else" branches.

**Path Coverage:** You can determine whether there are any scenarios that your test cases do not cover by looking at the various pathways that a program's code could take. Are there particular combinations of situations, for instance, that your tests do not examine?

Q5.

1. The problem is in the following code:

elif char.isnumeric():

output\_str += char \* 2

The flaw is that the output\_str duplicates (multiplies by 2) all numeric characters it encounters. The problem statement states that numeric characters should not change, hence this behaviour is improper.

Inspection of the Code: It is evident from a review of the code that the problem is with the elif char.isnumeric() condition.

Static Analysis: Examining the code shows that this condition has to be modified from if char.isdigit() to elif char.isdigit() in order to recognise numeric characters correctly and preserve their original state.

Manual Testing: You can also verify this by executing the code while using a variety of input values, including numeric characters, and seeing what happens when something goes wrong.