**COSC 3P95**

**Assignment #1**

**Fahad Khan, 6688998**

**October 9th, 2023**

1. In software analysis, the main differences between “sound” and complete” analysis is that sound analysis primarily refers to the accuracy and reliability when analysing software. It consistently delivers accurate results, avoiding any false positives or false positives. Complete analysis on the other hand, focuses on the further thoroughness of the software analysis. It will always report all bugs and vulnerabilities that exist in a software. Although there is extensive thoroughness, it will guarantee that false negatives are avoided, but may allow false positives.

**True Positive:** In software analysis, true positive is applied in a situation where a relevant scenario or condition in a software is appropriately identified as an issue.

**True Negative:** In software analysis, true negative is applied in a situation where a relevant scenario or condition in a software is appropriately identified as not an issue.

**False Positive:** In software analysis, false positive is applied when a relevant situation incorrectly identifies something as an issue when it is not actually an issue in the software at all.

**False Negative:** In software analysis, false negative is applied when a relevant situation incorrectly identifies something as not an issue, when there is an existing issue in the software.

**When "positive" means finding a bug:**

* True Positive: It would refer to instances where the analysis correctly identifies something as a bug when it is indeed a bug in the software.
* True Negative: It would refer to instances where the analysis correctly identifies something as not a bug when it is indeed not a bug in the software.
* False Positive: It would refer to instances where the analysis incorrectly identifies something as a bug when it is not a bug in the software.
* False Negative: It would refer instances where the analysis fails to identify something as a bug when it is indeed a bug in the software.

**When "positive" means not finding a bug:**

* True Positive: It would refer to instances where the analysis correctly identifies something as not a bug when it's indeed not a bug in the software.
* True Negative: It would refer to instance where the analysis correctly identifies something as a bug when it’s indeed not a bug in the software.
* False Positive: It would refer to instances where the analysis incorrectly identifies something as a bug when there is no bug in the software.
* False Negative: It would refer to instances where the analysis incorrectly identifies something as not a bug when it is indeed a bug in the software.

2. Code can be accessed in folder
3. test-case -> input | output | test-result

input -> "["integer-list"]"

integer-list -> integer | integer "," integer-list

integer -> random-integer

random-integer -> 0 | 1 | 2 | ... | 100

output -> "["sorted-integer-list"]"

sorted-integer-list -> integer | integer "," sorted-integer-list

test-result -> sorted-integer

2. Diagram image can be accessed in folder

* Starting with the content inside the filterData function, it takes in three parameters: ‘data’, ‘limit’, and ‘exceptions’. ‘Data’ refers to the list of integers, ‘limit’ refers to the integer values, and ‘exceptions’ refers to the list of exceptions.
* The first step is to generate random inputs of data for testing. This involves generating random lists for both data and exceptions and random integer values for the limit. When we are doing this, we need to make sure that the generated inputs cover a wide range of possible values.
* The second step is to execute random input testing. Execute the filterData function with the random inputs that were generated. Pass the randomly generated data, limit, and exceptions as input parameters.
* The third step is to observe and monitor the output of the function. Observe the executional output from the filterData function and verify that the function returned the expected output and results.
* The fourth step is to record or check for any possible errors and unexpected behaviour. Examine the code and output thoroughly to verify or track any errors, exceptions, or unexpected behaviour.
* The fifth step is to log and report the results and overall behaviour of the function. Log the input data used in the testing process and the results of the filterData function. This also includes reporting any errors, exceptions, or unexpected behavior observed.
* The sixth step is performing boundary testing. This means that we try to attempt on surpassing the input boundaries for those test cases where the limit is at its minimum, maximum, or within the boundaries. This can also mean that we have the flexibility to test with empty lists, containing single and negative elements.
* The seventh step is to repeat steps one to six, but with a larger number of randomly generated inputs instead. The greater the number of iterations we perform, the higher the chance of us discovering edge cases and unexpected behavior.



**Test Case #1: 30% Code Coverage**

This is an example of a basic test case with a list of basic values for data, limit, and exceptions and straightforward conditions.

data = [2, 3, 6, 9, 12]

limit = 8

exceptions = [12]

Expected Output:

[0.25, 0.375, 0.75, 18, ‘12\_EXCEPTION’]

**Test Case #2: 60% Code Coverage**

This is an example of a test case where the boundaries are tested, setting at least one value of each parameter to 0.

data = [5, 1, 0, 4]

limit = 0

exceptions = [5, 0]

Expected Output:

[‘5\_EXCEPTION’, 1, ‘0\_EXCEPTION’, 8]

**Test Case #3: 90% Code Coverage**

This is an example of a more complex test case that consists of input data that are in various conditions.

data = [7, 11, 10, 14, 16]

limit = 12

exceptions = [10]

Expected Output:

[0.5833, 0.9166, ‘10\_EXCEPTION’, 28, 32]

**Test Case #4: 100% Code Coverage**

This is an example of a more comprehensive test case as it aims to achieve every single branch to ensure the function is functional in every possible situation.

data = [2, 4, 6]

limit = 4

exceptions = []

Expected Output:

[0.6666, 1, 12]



**Mutated Version #1: Change while loop to for loop**

def filterData(data, limit, exceptions):

filtered\_data = []

for item in data:

if item in exceptions:

…

**Mutated Version #2: Remove closing quotation in line 7**

def filterData(data, limit, exceptions):

…

Modified\_item = item + “\_EXCEPTION

…

**Mutated Version #3: Change the order of the if statements**

def filterData(data, limit, exceptions):

…

if item > limit

modified\_item = item \* 2

elif item in exceptions:

modified\_item = item + “\_EXCEPTION”

…

**Mutated Version #4: Change the operator sign to addition in line 9**

def filterData(data, limit, exceptions):

…

if item > limit

modified\_item = item + 2

…

**Mutated Version #5: Add an extra if condition while index < len(data)**

def filterData(data, limit, exceptions):

…

while index < len(data):

item = data[index]

if item == limit:

modified\_item = item + "\_EQUAL\_LIMIT"

elif item in exceptions:

modified\_item = item + "\_EXCEPTION"

elif item > limit:

modified\_item = item \* 2

else:

modified\_item = item / limit

…

**Mutated Version #6: Change exception condition from ‘if item in exceptions’ to ‘if item not in exceptions’**

def filterData(data, limit, exceptions):

…

if item not in exceptions:

modified\_item = item + “\_NOT\_EXCEPTION”

…

1. **Test Cases ranked from best to worst:**
2. Test Case #4: This test case is designed to achieve 100% code coverage, which guarantees that every branch and code line is gone through during testing. Additionally, the test case is very effective in detecting any sort of change based off the mutated versions. This implies that the test case is very likely to detect any possible errors or bugs.
3. Test Case #3: This test case is designed to achieve 90% code coverage, which covers majority of the branches and lines, but of-course not everything. It is effective with detecting any sort of change based off the mutated versions through most parts of the code. Overall, it is a fairly good test case in terms of testing the code behaviour, but it covers less code than Test Case 4.
4. Test Case #2: This test case is designed to achieve 60% code coverage, which mainly covers boundary testing and cases where all parameters contain a value of 0. This test case is of-course useful when it comes to detecting some errors based off mutated versions within the function conditions. However, it is more inefficient compared to Test Cases 3 and 4 as it covers less branches and lines of code.
5. Test Case #1: This test case is designed to achieve 30% code coverage, which covers a basic list of values for the parameters and offers minimal conditions. While it does cover some bits of the code, it does not a good job in exploring the overall code behaviour in more complex situations and out-of-boundary conditions. This test case covers the least amount of code out all test cases which is why it’s ranked the last.

**Path Static Analysis:**

This type of analysis focuses on the evaluation of all possible execution paths through code program. It offers a more comprehensive analysis as it involves further examination of the various conditions and branches. For the code above specifically, we need to manually identify all possible paths within the code, considering all combinations on the branches and conditions. Examples can be whether an item is greater than or less than a limit, if an item is in exceptions, etc. Next step would be to determine which paths are executable or not depending on the inputs entered. Finally, make an analysis of the effect of the various paths discovered.

**Branch Static Analysis:**

This type of analysis focuses on the various branches and conditional statements that are applied within the code. For instance, the code above consists of if, elif, and else statements which are things branch static analysis observes. The first step is to of-course identify the conditions and branches and determine whether they cover all possible situations based on the input data. Once identified, we need to verify whether each condition and branch was well executed in every possible situation. Lastly, we analyze everything we gathered from the last two steps; ultimately, we examine on the number of conditions that were executed within different inputs of data to understand the overall coverage of the code.

**Statement Static Analysis:**

This type of analysis focuses on individual lines of code. For this analysis, we would want to examine every single line of code to determine whether if there are any possible errors or issues within those lines. For example, if there is a syntax error in line 7, where supposedly the closing quotation was missing after EXCEPTION, the analysis will record that. Once we are done checking for any other syntax errors, we then move on to assuring that all parameters (data, limit, exceptions) are defined and applied correctly. Next, we need to examine conditions such as while loops, if/elif statements, and the overall logical implementation to see if they are also applied correctly.

2. From the code snippet provided, there seems to be a bug located in line 7, where the line is output\_str + = char \* 2. The problem with this line is that it doubles the character that is inputted, whether if it is a numeric character or not. One of the requirements for this code was to keep the numeric characters unchanged. If the input is a numeric character, line 7 can simply double that character, changing its value which is against the code requirements. For example, if the value 5 is inputted into the code, the value will double and become 10 which is incorrect, it should remain as the value 5.

**Identifying the bug using:**

Manual Random Testing: We can further verify the fact that line 7 is a bug by applying combinations of input values which will include uppercase, lowercase, and numeric values to observe the output behaviour. For example, if the input was ‘Hello my name is Fahad 22’, the output is expected to be ‘hELLO MY NAME IS fAHAD 22’, but if the bug is still present, the output will be ‘hELLO MY NAME IS fAHAD 44’. When this output is generated, it will be obvious to determine where the issue lies within the code as we know the value ‘44’ is incorrect and the correct value should be ‘22’.