COSC 3P95- Software Analysis & Testing

Assignment 1

Name: Michael Boulos **Student ID:** 6973523

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 is the detection of vulnerabilities and bugs in software analysis that does report any false detections; essentially, a "sound" analysis technique only ever reports bugs that are absolutely present in the software system. "Complete" analysis is the detection of all possible vulnerabilities and bugs in software analysis, preventing a bug going undiscovered; if an analysis technique is "complete", it will report all actual bugs in the system, but may or may not also report some areas of the system without bugs.

- True Positive: Reporting a positive detection for some test, while the true value is positive.
- True Negative: Reporting a negative detection for some test, while the true value is negative.
- False Positive: Reporting a positive detection for some test, while the true value is negative.
- False Negative: Reporting a negative detection for some test, while the true value is positive.

Based on the usage of these terms: if "positive" means finding a bug, then a "sound" analysis would not report any false positives and a "complete" analysis would prevent false negatives. If "positive" means not finding a bug, then a "sound" analysis would not report any false negatives and a "complete" analysis would prevent false positives.

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.).

My method for generating test-cases and evaluating them is by generating 50, 20 integer arrays, where the integers are from -2000 to 2000, mind you, these values are changeable and the tester/developer can change how many test-cases are generated and what size per case. The sorting algorithm I implemented is a sorting algorithm 'mutation sort', which sorts a given array of integer using the built in Arrays.sort() method, with the modification that there is a small chance of switching the places of two integers in the array, a mutation, to have a variety of passing and failing test-cases. Per test-case, a 'status' value is logged to illustrate whether the current run of a test case had resulted in a successful sort or a failing sort, by comparing the sorted array with the array given by just calling Arrays.sort() on the original unsorted array.

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

To generate a list of integers to sort, in the case of the test-case generator above inclusively between -2000 and 2000, use the context free grammar:

C -> integer, C | integer | ϵ {where 'integer' is an integer >= -2000 and <= 2000}

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

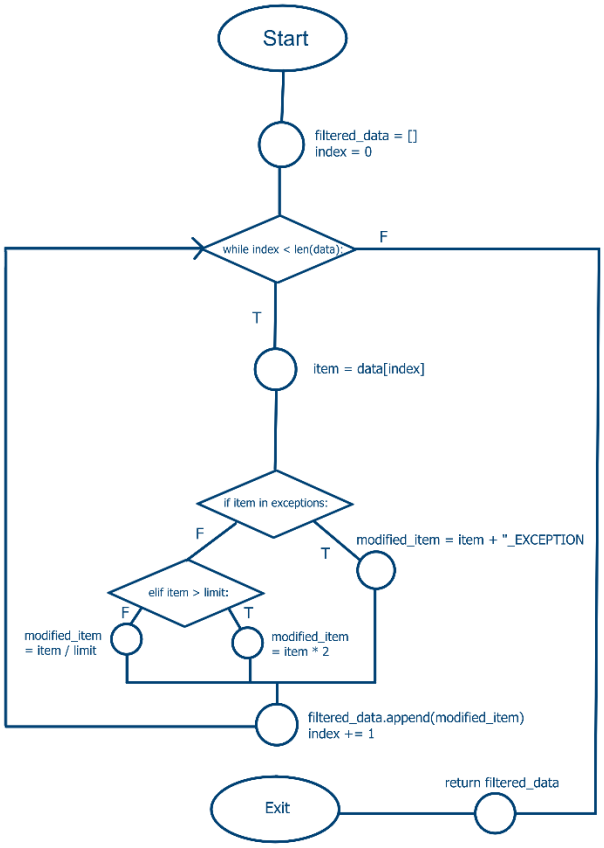
```
def filterData(data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
        item = data[index]
        if item in exceptions:
          modified_item = item + "_EXCEPTION
        elif item > limit:
          modified_item = item * 2
        else:
         modified_item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

To apply “random testing with feedback” to the given code, some important specifications would need to be considered to create a viable test harness for the above algorithm. The harness would need to be developed to include a helper function to generate random data values of the same type in the form of the array/list ‘data’, a limit variable which is comparable to the type of values within the array ‘data’, and an ‘exceptions’ array/list with the same type of value(s) in the previous two variables that provides the filtering aspect of the algorithm. Once all those values are generated for numerous test cases of possible inputs, the test cases should be stored and undergo some mutation step, where their values are changed and mutated. The generated and mutated test cases are then run through the algorithm, where their given outputs are judged by some user-decided metric (depending on what kind of code coverage the developer would like to test more on), determining which test cases are reused into the harness to guide subsequent inputs. The reason I suggest the “random testing with feedback” method is due to better guide the testing creation, as well to be more applicable to multiple types of input data given the algorithm just accepts ‘data’ as an input.

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.

Case 1: filterData([], 1, [1,2])

Code Coverage: 4/13 -> 0.308 = 30.8%

Case 2: filterData([2,4], 2, [2,4])

Code Coverage: 9/13 -> 0.692 = 69.2%

Case 3: filterData([5,7], 5, [5])

Code Coverage: 11/13 -> 0.846 = 84.6%

Case 4: filterData([10,20,30], 25, [20])

Code Coverage: 13/13 -> 1 = 100%

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

Mutant 1 (Arithmetic mutations) :

```
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
```

Mutant 2 (Statement mutations) :

```
def filterData(data, limit, exceptions):
filtered_data = []
index = 1
while index < len(data):
        item = data[index]
        if item in exceptions:
          modified_item = item + "_EXCEPTION
        elif item > limit:
          modified_item = item
        else:
         modified_item = item / limit

        filtered_data.append(modified_item)
        index += 1

    return filtered_data
```

Mutant 3 (Variable mutations) :

```
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 / index

        filtered_data.append(limit)
        index += 1

return filtered_data
```

Mutant 4 (Arithmetic and Statement mutations) :

```
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 += 2

return filtered_data
```

Mutant 5 (Arithmetic and Variable mutations) :

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

        filtered_data.append(modified_item)
        index += 1
```

```
return filtered_data
```

Mutant 6 (Arithmetic, Statement and Variable mutations) :

```
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 = index / 2
        else:
         limit = item / limit

        filtered_data.append(modified_item)
        index += 1

return filtered_data + data
```

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 1 assert: filterData([], 1, [1,2]) == []	Test 2 assert: filterData([2,4], 2, [2,4]) == ['2_EXCEPTION', '4_EXCEPTION']	Test 3 assert: filterData([5,7], 5, [5]) == ['5_EXCEPTION', 14]	Test 4 assert: filterData([10,20,30], 25, [20]) == [0.4, '20_EXCEPTION', 60]
Mutant 1 (Arithmetic mutations)	output: [] PASSED	output: ['2_EXCEPTION', '4_EXCEPTION'] PASSED	output: ['5_EXCEPTION', 3.5] FAILED	output: [250, '20_EXCEPTION', 15] FAILED
Mutant 2 (Statement mutations)	output: [] PASSED	output: ['4_EXCEPTION'] FAILED	output: [7] FAILED	output: ['20_EXCEPTION', 30] FAILED
Mutant 3 (Variable mutations)	output: [] PASSED	output: [2, 2] FAILED	output: [5, 5] FAILED	output: triggered error FAILED
Mutant 4 (Arithmetic and Statement mutations	output: [] PASSED	output: ['2_EXCEPTION'] FAILED	output: ['5_EXCEPTION'] FAILED	output: [-15, 60] FAILED
Mutant 5 (Arithmetic and Variable mutations)	output: [] PASSED	output: ['2_EXCEPTION', '2_EXCEPTION'] FAILED	output: ['5_EXCEPTION', 14] PASSED	output: [0, '25_EXCEPTION', 60] FAILED

Mutant 6 (Arithmetic, Statement and Variable mutations)	output: [] PASSED	output: ['2_EXCEPTION', '4_EXCEPTION', 2, 4] FAILED	output: ['5_EXCEPTION', 0.5, 5, 7] FAILED	output: triggered error FAILED

The calculated ranking of test-cases (best -> worst) are: Test 4, Test 2, Test 3, Test 1;

the ranking of these test-cases are based on a calculated mutation_score, which is calculated by the formula of killed_mutants / total_mutants (comparing how many mutants gave a different output than the expected as compared to the total number of mutants). Since Test 4 killed all of its tested mutants, it is ranked the highest, followed by Test 2 and Test 3 – which killed the same amount of mutants, and Test 1, which killed no mutants.

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

To analyse the above code using static analysis, we would have to go through the source code itself, without directly testing the code by running it or producing a test program and using that – albiet parsers that automate this analysis can still be used on the source code. This means that in order to better understand the code and evaluate it, we would go through the individual paths within the source code to understand the steps by which resources are changed and initialized through the lifetime of the code to check for resource considerations, how data is handled, allocated, and freed throughout, and whether or not any specific execution path may never logically be reached within the domain of input. Branch analysis would be used to specifically target an evaluation of the complexity of the branches within the code, speaking to maintainability, finding of bugs related to a specific branch of code, and finding any possible unconsidered edge cases of valid input. Statement analysis would tackle the specific statements/lines of code to determine the possible side-effects and outcomes of a particular statement.

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, based on the specified purpose of the code as outlined, it can be identified that there is a bug that outputs numerical characters in the original string as multiplied by 2 in the processed output string. This occurs in line 7, “output_str += char * 2”; there is a bug since the original specifications of the code is that numerical characters should be added to the output as they are, unchanged. By using additional analysis by testing on the code, it can be identified that the code, when a numerical character is encountered, adds 2 instances of that character in the output string, ie. 1a2b3c becomes 11A22B33C, rather than leaving that numerical character unchanged.

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” -> 1
    ii. “CCDDEExy” -> Does not trigger bug
    iii. “1234567b” -> 1
    iv. “8665” -> 8

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

My delta debugging algorithm works by taking a given input string, checking if there exists a bug when in the output when processing the string with the algorithm above, this is done through a function which takes in a string and checks for duplicate consecutive integers (since that is what the discovered bug causes). If there exists a bug in the given string, it is split into two halves, the substrings are then checked for bugs; if at any point, a processed substring lacks the bug, the function will return and we will not consider any substring made from that string. The function only ends when the first instance of the smallest possible substring that reveals the bug is found, which is assigned as the minimal bug string, this string is then printed along with the original input string. By finding this minimal bug string, we can essentially better pinpoint which characters in the string are causing the bug – therefore making debugging more focused and programmatically driven.

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)**

Github link: https://github.com/MikeBlu/COSC3P95-A1_MB

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***