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: George Barakat **Student ID:** 6429161

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)**

Answer:

A sound analysis technique will never report a bug or vulnerability that does not actually exist in a software. When soundness concludes some fact about the program, it is indeed true in all runs of the program. Soundness prevents false positives.
Soundness guarantees that there will be no false positive but cannot guarantee false negatives (It does not guarantee that it will find all the vulnerabilities that exist).

A complete analysis technique will always report all bugs and vulnerabilities that exist in the software. A complete analysis technique is one where if there is a bug, it will be reported. Completeness prevents false negative. Completeness guarantees that there will be no false negative but cannot guarantee false positives (It does not guarantee that the technique will not report a vulnerability that does not exist).

- When positive means finding a bug:

True positive: Occurs when a tool correctly identifies a bug that exists in the code.

True negative: Occurs when a tool correctly identifies that there is no bug in the code.

False positive: Occurs when a tool incorrectly reports a vulnerability or bug that does not actually exist in the code.

False negative: Occurs when a tool fails to report a vulnerability or bug that exists in the code.

- When positive means not finding a bug:

True positive: Occurs when a tool correctly identifies that there is no bug in the code.

True negative: Occurs when a tool correctly identifies a bug in the code.

False positive: Occurs when a tool incorrectly identifies a bug-free code as having a bug.

False negative: Occurs when a tool fails to identify or report a bug in a code where no bugs are believed to exist, but there actually is one.

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)**

Answer for Q2 is in a separate file.

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)**

Answer:

- For this program I would generate random inputs, a list containing random elements of different types like (integers, floats, strings). I would also make sure to include zero and negative values, to see how the program behaves with different types of integers.
  Taking exceptions into consideration is important too, I would do that by creating lists of random elements that may or may not be in the data list.

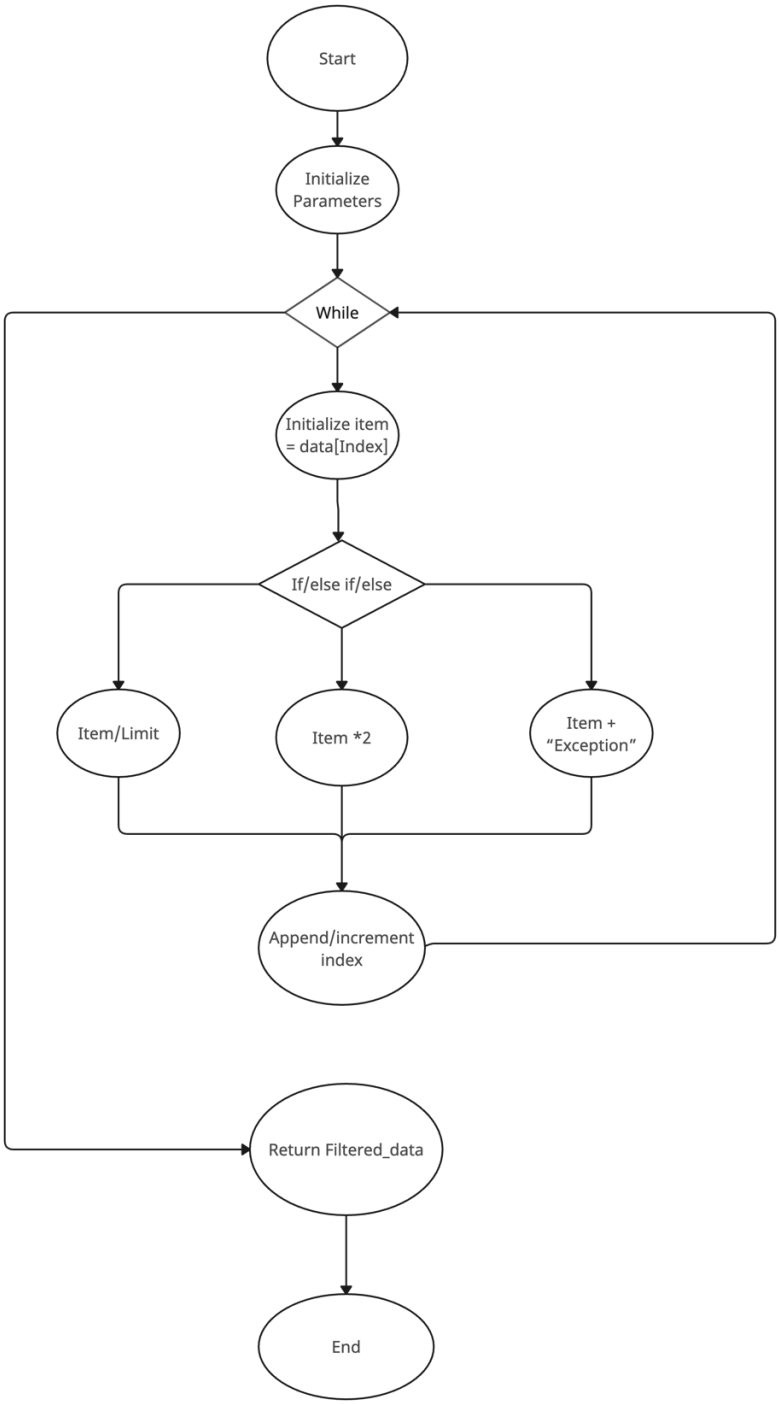
- Then I would define the expected based on the limits and exceptions we have. For example: when item is exceptions, append “_EXCEPTION”. When the item is greater than limit, double it. Otherwise, divide it by 2.

- Then I would test the execution by comparing the actual output with the expected output.

- After that I will evaluate the results, if actual output matches the expected output, then the test has passed, if it doesn’t match then the test has failed, and we need to debug the code.

- I would iterate the test case 100 – 1,000 times, taking into consideration edge cases like (empty lists, large and small values, list that is full of exception cases)

A) Control Flow Graph



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**

```
List<String> data = Arrays.asList("2", "3", "2");
int limit = 4;
List<String> exceptions = Arrays.asList("2", "3");
List<String> result = filter.filterData(data, limit, exceptions);
assertEquals(Arrays.asList("2_EXCEPTION", "3_EXCEPTION", "2_EXCEPTION"), result);
```

This is the first condition inside the loop where the item is an exception. **(Code Coverage 30%)**

- **Test Case 2**

```
List<Integer> data = Arrays.asList("6", "7", "8");
int limit = 3;
List<Integer> exceptions = Arrays.asList("0");
List<Integer> result = filter.filterData(data, limit, exceptions);
List<Integer> expected(Arrays.asList("12", "14", "16"), result);
```

This is the second condition inside the loop where the item is greater than the limit. **(Code Coverage 40-50%)**

- **Test Case 3**

```
List<Integer> data = Arrays.asList("1", "2", "4");
int limit = 6;
List<Integer> exceptions = Arrays.asList("2");
List<Integer> result = filter.filterData(data, limit, exceptions);
List<Integer> expected(Arrays.asList(1 / 6, "2_EXCEPTION", 4 / 6), result);
```

This condition inside the loop where the item is smaller than the limit which is the last condition, and the loop has iterated through all conditions. **(Code Coverage 70-80%)**

- **Test Case 4**

```
List<Integer> data = Arrays.asList(5, 10, 20, 25, 30);
int limit = 20;
List<Integer> exceptions = Arrays.asList(10, 25);
List<Integer> result = filter.filterData(data, limit, exceptions);
List<Integer> expected = Arrays.asList(5 / 20.0, "10_EXCEPTION", 1 , 50, 60);
```

This will cover all the conditions and branches present in the filterData (item in exceptions list, item greater, less, and equal to the limit). **(Code Coverage 100%)**

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

- **Mutated Code 1**

```
elif item > limit:
        modified_item = item + 2
```

Changed the elif block from multiplication to addition.

- **Mutated Code 2**

```
elif item < limit:
        modified_item = item * 2
```

Changed the logical operator from > to <.

- **Mutated Code 3**

```
 index += 2
```

Changed the increment value of the loop counter index.

- **Mutate Code 4**

```
if item not in exceptions:
        modified_item = item + "_EXCEPTION"
```

Negated the condition that checks if the item is in exceptions.

- **Mutate Code 5**

```
else:
        # modified_item = item / limit          (Commented/Cancelled)
        filtered_data.append(item / limit)
```

Modified the item being appended to filtered_data without using modified_item.

- **Mutate Code 6**

```
# if item in exceptions:                        (Whole block is commented/cancelled)
    #    modified_item = item + "_EXCEPTION"
    # elif item > limit:
```

Remove the handling of exceptions and treat normal.

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 can detect Mutated Code 4 (because the items 2 and 3 are both in exceptions and they get the _EXCEPTION condition, which means we should expect (2, 3, 2) as an answer.

- Test Case 2 can detect Mutated Code 1 (items are multiplied by 2 when they should be added by 2) (Expected answer would be: 8, 9, 10) and can also detect Mutated Code 2 (items are being multiplied, which shouldn't happen with the given limit).

- Test 3 can detect Mutated Code 1 because the modified code would incorrectly add 2 to the item 4, instead of dividing it. Can detect Mutated Code 2 because the modified code would incorrectly multiply the item 4, which is greater than the limit, instead of dividing it. Can detect Mutated Code 4 because "2" is in exceptions. Can detect Mutated Code 5 due to the difference in the division result. Can detect Mutated Code 6 since the item 2 don't get the _EXCEPTION condition.

- Test Case 4 can an detect Mutated Code 1, 2, and 3 (due to the change in index increment, some values won't be processed), can detect Mutated Code 4 (as 10 and 25 are in exceptions and get _EXCEPTION condition). Can detect Mutated Code 5 due to the difference in the division result. Can detects Mutated Code 6 (10 and 25 don't get _EXCEPTION condition). This test can detect all mutated codes.

  Ranking:

  1. Test Case 4 (coverage 100%): Most effective and most likely detects all mutations.
  2. Test Case 3 (Coverage 80%): This test case becomes more effective and can detect most mutations except Mutated Code 3.
  3. Test Case 2 (coverage 50%): Detects fewer mutations than Test Case 3 and 4
  4. Test Case 1 (coverage 30%): Is the least effective as it mainly detects the exception handling part.

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

- **Path:** I would use path to test all possible paths in the code considering sequence of branches. Such as an item in the exception followed by an item that is greater than limit or Item is greater than limit followed by an item that falls in the else block.

- **Branch:** I would make sure that each branch of every if statement (if, elif, else) is being executed. Such as a test case where item is in exceptions, item is greater than limit, or neither of conditions are true. Another test case for while loop to make sure it iterates multiple times or doesn't iterate at all (empty data)

- **Statement Static Analysis:** I would read the code to ensure that every statement in the code is being executed at least once. I would read through each of the while loop conditions, assume inputs when item is in exceptions, item is greater, smaller, and equal to the limit.

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.

- By manually looking at the code I can identify two bugs.

1- Line "output_str += char *2", the numeric character is being doubled and added twice to the output_str, when numeric characters are supposed to stay unchanged based on the program requirements.

2- The function is missing a case to convert lowercase characters to uppercase. (Requirements say switch uppercase characters to their lowercase counterparts and vice versa)

Code should be added:

```
elif char.islower()
output_str += char.upper())
```

3- If input contains any character other than alphabet they will also be switched to upper case.
Line causing it (output_str += char.upper()).

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)**

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***