Assignment 1

Gideon Oludeyi (7333586)

October 16, 2023

1 Question 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)**

1. Sound vs Complete

   A sound analysis does not report any *false positives.* In other words, it does not claim that a piece of code has a vulnerability when it actually doesn't.

   A complete analysis does not report any *false negatives.* This means that it will detect/report all possible vulnerabilities.

   Although a sound analysis can confidently identify a vulnerability, a complete analysis can detect *all* vulnerabilities in the system. This means that a sound analysis can fail to detect *all* vulnerabilities, and a complete analysis can sometimes flag a piece of code that is not a vulnerability.

2. True Positive - when an analysis tool reports the positive case, and the actual outcome is indeed positive.

   Positive: finding a bug - when the system has a bug, and the analysis tool correctly identifies that it has that bug.

   Positive: *not* finding a bug - when the system does not have a bug, and the analysis tool correctly claims that the system does not have a bug.

3. True Negative - when an analysis tool reports the negative case, and the actual outcome is the negative case.

   Positive: finding a bug - when the system does not have a bug, and the analysis reports that there is no bug.

   Positive: *not* finding a bug - when the system has a bug, and the analysis reports that the bug exists.

4. False Positive - when an analysis tool reports the positive case, but the actual outcome is negative.

   Positive: finding a bug - when the system does not have a bug, but the analysis tool reports a bug.

   Positive: *not* finding a bug - when the system has a bug, but the analysis tool does not report it.

5. False Negative - when an analysis tool reports the negative case, but the actual outcome is positive.

   Positive: finding a bug - when the system has a bug, but the analysis tool reports that there is not bug.

   Positive: *not* finding a bug - when the system does not have a bug, but the analysis tool reports that a bug exists.

2 Question 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.

2.1 A)

Your submission should consist of:

1. Source code files for the sorting algorithm and the random test case generator.

   Source Code: `q2.py`

2. 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.

3. Comments within the code for better understanding of the code.

4. Instructions for compiling and running your code.

   1. Ensure you have `python` installed

   2. Run `python q2.py` to run the random test case generator multiple times. By default it will perform 30 random test case executions. But you can run `python q2.py 10` to run it 10x or however many times.

5. Logs generated by the print statements, capturing both input array, output arrays for each run of the program.

   Log file: `q2n30.txt`

6. 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.).

   Log file: `q2n30.txt`

Explanation

The random test case generator uses the 3rd generation random testing strategy. It randomly selects from a list of predefined test cases and runs the sorting algorithm on the input. Depending on the sorting algorithm's output (if it fails to match the output of the built-in `sorted` function in Python), we generate new test cases that are mutations of the original test case. And the next iteration of the test case randomly selects out of the increased pool of test cases (including the mutations), repeating the cycle.

The mutation transformations that were applied on failing inputs were the following:

- Swap - for every consecutive pair of unordered elements, generate a new test case where the order of the consecutive pair corrected. e.g. given a list $[..., 23, -43, ...]$, we generate a mutated list $[..., -43, 23, ...]$ where the pair is ordered, while leaving all other elements in their original position

- Extend - generate a new test case by appending a random number of random integers to the original test case.

This method was able to identify a bug due to a condition in the `while` loop of the `sort` function: the first condition in the while loop should have been `i >= 1` instead of `i > 1`. The `>` alone prevents the first element of the input list from being sorted with the rest of the list. This bug results in wrong outputs where the first element always remains in the same place, even though the rest of the list is sorted.

This pattern can be observed in the `q2n30.txt` logs. For all the failed inputs, the difference between the expected and actual outputs was that the first integer (23) still remained in the first position.

```
Pass:  input was [-43, 23]; output was [-43, 23]
```

```
   Pass:  input was [7]; output was [7]
   Fail:  input was [23, -43]; expected [-43, 23]; but got [23, -43]
   Pass:  input was []; output was []
   Fail:  input was [23, -43, -99, -49, 58, -92]; expected [-99, -92, -49, -43, 23, 58];
but got [23, -99, -92, -49, -43, 58]
   Fail:  input was [23, -43, -40, -21, 3]; expected [-43, -40, -21, 3, 23]; but got [23,
-43, -40, -21, 3]
   Pass:  input was [-43, 23]; output was [-43, 23]
   Fail:  input was [23, -99, -43, -49, 58, -92]; expected [-99, -92, -49, -43, 23, 58];
but got [23, -99, -92, -49, -43, 58]
   Fail:  input was [23, -43, -99, -49, -92, 58]; expected [-99, -92, -49, -43, 23, 58];
but got [23, -99, -92, -49, -43, 58]
   .
   .
   .
```

2.2 B)

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

```
   TestCase →   "[" List "]" | "[]"
   List     →   Number | List ", " Number
   Number   →   PosInt | "-" PosInt
   PosInt   →   Digit  | Digit PosInt
   Digit    →   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

3 Question 3

3.1 A)

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

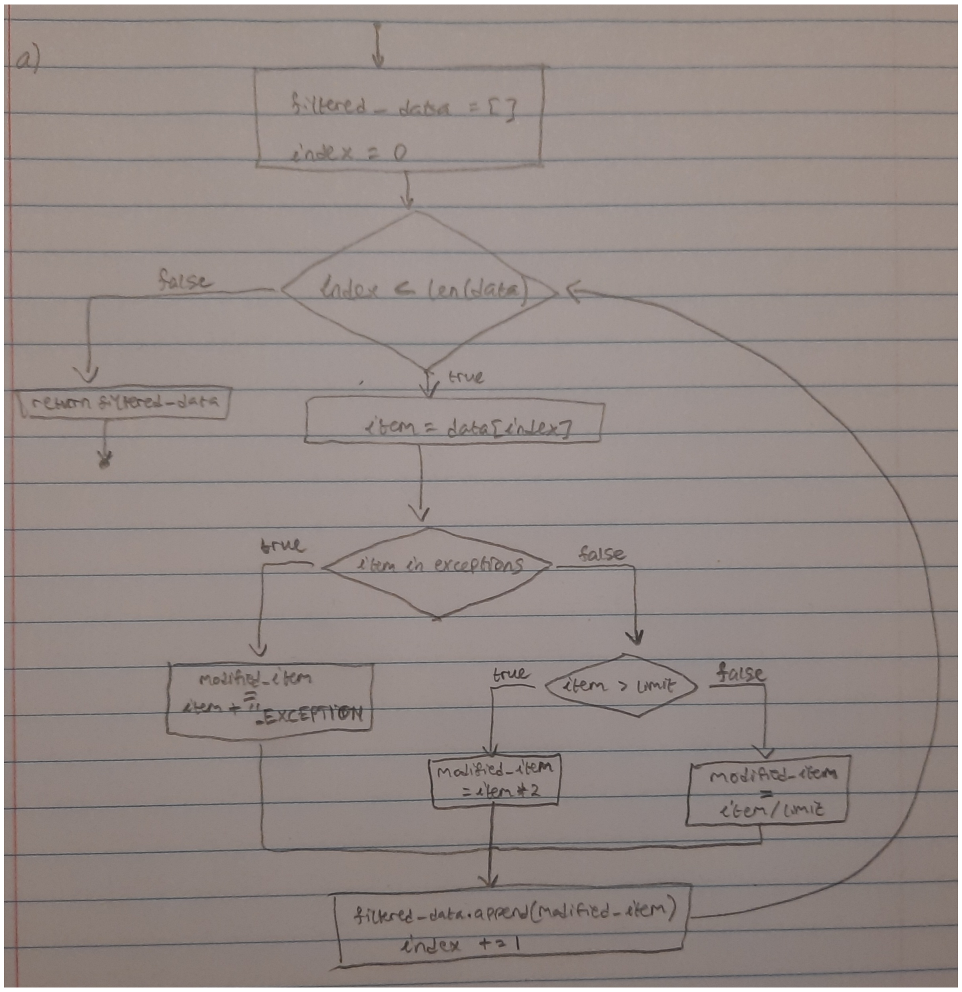
```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item in exceptions:
            modified_item = item + "_EXCEPTION
        elif item > limit:
            modified_item = item * 2
        else:
      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



3.2 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)**

4 Question 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.

Lines of Code: 13

1.

```
def test_case_1():
    data = [99]
    limit = 5
    exceptions = [99]

    output = filterData(data, limit, exceptions)

    assert(output == ["99_EXCEPTION"])
```

Statement Coverage: 9/13 = 69.23%

Branch Coverage: 1/3 = 33.33%

2.

```
def test_case_2():
    data = [4]
    limit = 5
    exceptions = []

    output = filterData(data, limit, exceptions)

    assert(output == [2])
```

Statement Coverage: 10/13 = 76.92%

Branch Coverage: 1/3 = 33.33%

3.

```
def test_case_3():
    data = [7]
    limit = 5
    exceptions = []

    output = filterData(data, limit, exceptions)

    assert(output == [14])
```

Statement Coverage: 10/13 = 76.92%

Branch Coverage: 1/3 = 33.33%

4.

```
def test_case_4():
    data = [5]
    limit = 5
    exceptions = []

    output = filterData(data, limit, exceptions)

    assert(output == [2.5])
```

Statement Coverage: 10/13 = 76.92%

Branch Coverage: 1/3 = 33.33%

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

1.

```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item not 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
```

2.

```
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
```

3.

```
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
```

4.

```
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
```

5.

```
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
```

6.

```
    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
```

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.

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

5 Question 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.

Description - the bug causes the output to duplicate all numeric characters in the input string. On **line** 7, `char * 2` is appended to `output_str` instead of appending just `char`, which causes the bug to surface as duplicate numeric characters in the output string. As a result, the output of the function for input `"a86"` is `"A8866"`, whereas it should be `"A86"`.

1. **Identification Strategy** - I was able to identify the bug by manually reviewing the code with the different types of input characters as specified in the requirement. First I walked through the function with an uppercase string `"A"`, which would satisfy the `char.isupper()` condition. After manually verifying that the body of the conditional statement produces the expected result `"a"`, I proceeded to try a numeric character. Using the string "8", I manually stepped through the code, and upon reaching **line** 6 `char.isnumeric()`, I examined the body of the conditional and discovered that `char * 2` is being appended to the output string, which duplicates the numeric character, rather than leaving it unchanged. Thus the output string would have been `"88"` instead of `"8"`. Therefore, I identified that **line** 7 was the source of the bug.

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)**

Source Code: `deltadebuggingQ5.py`

1. The algorithm takes two inputs: the input string that is the cause of the bug, and the expected output of the calling `processString` on the input.

2. This is a recursive algorithm, so we first define the base case, which is when the input string does not produce unexpected output. In this case, we return `None`, specifying that we cannot find any smaller input that can cause failure.

3. We also define another base case, which is when the length of the input string is `0` or `1`. In this case, since the function still fails on this input, it is the smallest string size we can have. Therefore, we return the input string as the minimal size found.

4. Then we follow the delta debugging algorithm by splitting the input string `delta` into equally-sized sub-strings $\Delta_1, \Delta_2, ..., \Delta_m$, and test the function on each Δ_i and its inverse $\nabla_i = \Delta_1 + \Delta_2 + ...\Delta_{i-1} + \Delta_{i+1} + \Delta_{i+2} + ...$

   When `processString()` fails on a Δ_i (i.e. output does not match expected output), I recursively call `test()` on Δ_i with $n = 2$ to possibly find a smaller string that produces the bug.

   When the function fails on a ∇_i (i.e. result does not match expected result), I recursively call `test()` on the ∇_i with $n = n - 1$.

   If none of the recursive `test()` calls on Δ_i and ∇_i cause the function to fail, then I recursively call `test()` on the current Δ instead with $n = n * 2$.

   If any of the `test()` invocations above produce a smaller string that causes the function to fail (i.e. output of `test()` is not `None`), then we return the smaller string. Otherwise, if none of the recursive calls to `test()` found a smaller substring, then we return the current Δ, since that is the smallest we have found.

6 Question 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)**