COSC 3P95 – Assignment #1

Chris Orr - #6755383

Monday, Oct. 16th, 2023

1. Sound analysis refers to the ability of an analysis technique or tool to not report any false positives. This means that an analysis which is fully sound will never report a bug that does not exist, only ones which do exist.

Completeness refers to the ability of an analysis technique or tool to detect all actual bugs and vulnerabilities that exist in a software. This means that an analysis which is fully complete will report on bugs which are found within the whole software.

The difference between completeness and soundness is that completeness will report on bugs within the whole software but can report false negatives (Finds bugs within the whole scope of the program but could report bugs that do not exist or are not important). Soundness means that the report is correct but may have missed bugs outside of the report's scope, resulting in the possibility of false negatives (Technique reports that there are no bugs, but there are bugs outside of the technique's scope).

- A **true positive** refers to a correct positive result in the test search. If the technique is searching for bugs, a true positive means that it has indeed found a bug.
- A **true negative** refers to a correct negative result in the test search. If the technique is searching for bugs, a true negative means that there are no bugs.
- A **false positive** refers to an incorrectly reported positive result in the test search. If the technique is searching for bugs, a false positive means that the technique has reported a bug exists when a bug does not exist.
- A **false negative** refers to an incorrectly reported negative result in the test search. If the technique is searching for bugs, a false negative means that the technique has reported that a bug does not exist when a bug does exist.

These terms change when there are different analysis technique goals, such as identifying a bug or not identifying a bug. If the technique is to not identify a bug, then:

- A true positive will refer to the test reporting that a bug does not exist when a bug does not exist.
- A true negative will refer to the test reporting that a bug does exist when a bug does exist.
- A false positive will refer to the test reporting that a bug does not exist when a bug does exist.
- A false negative will refer to the test reporting that a bug does exist when a bug does not exist.

2. My code generates a number of arrays of random length with random values. It then sorts these arrays with the Bubble Sort Algorithm and prints original input arrays and the sorted arrays to console. If the sorting algorithm returns an incorrectly sorted array, the log statements will show that the test did not pass or can throw an exception.

The issues covered by my test cases are:

- Incorrect sorting due to erroneous logic, such as *"arr[j] < arr[j + 1]"* in the Bubble Sort Algorithm. The arrays will print out sorted incorrectly and the verifySorted() method will return false for each incorrectly sorted array. In console, the input array is compared to the incorrectly sorted array and a message reads: "Is Sorted Correctly: False".

- ArrayOutOfBoundsException error - when an array out of bounds exception is introduced in the code such as *"i < length + 1"* instead of *"i < length"* in the generateRandomArray loop, the exception is printed to console, alerting the programmer that there is an incorrect loop somewhere in the code.

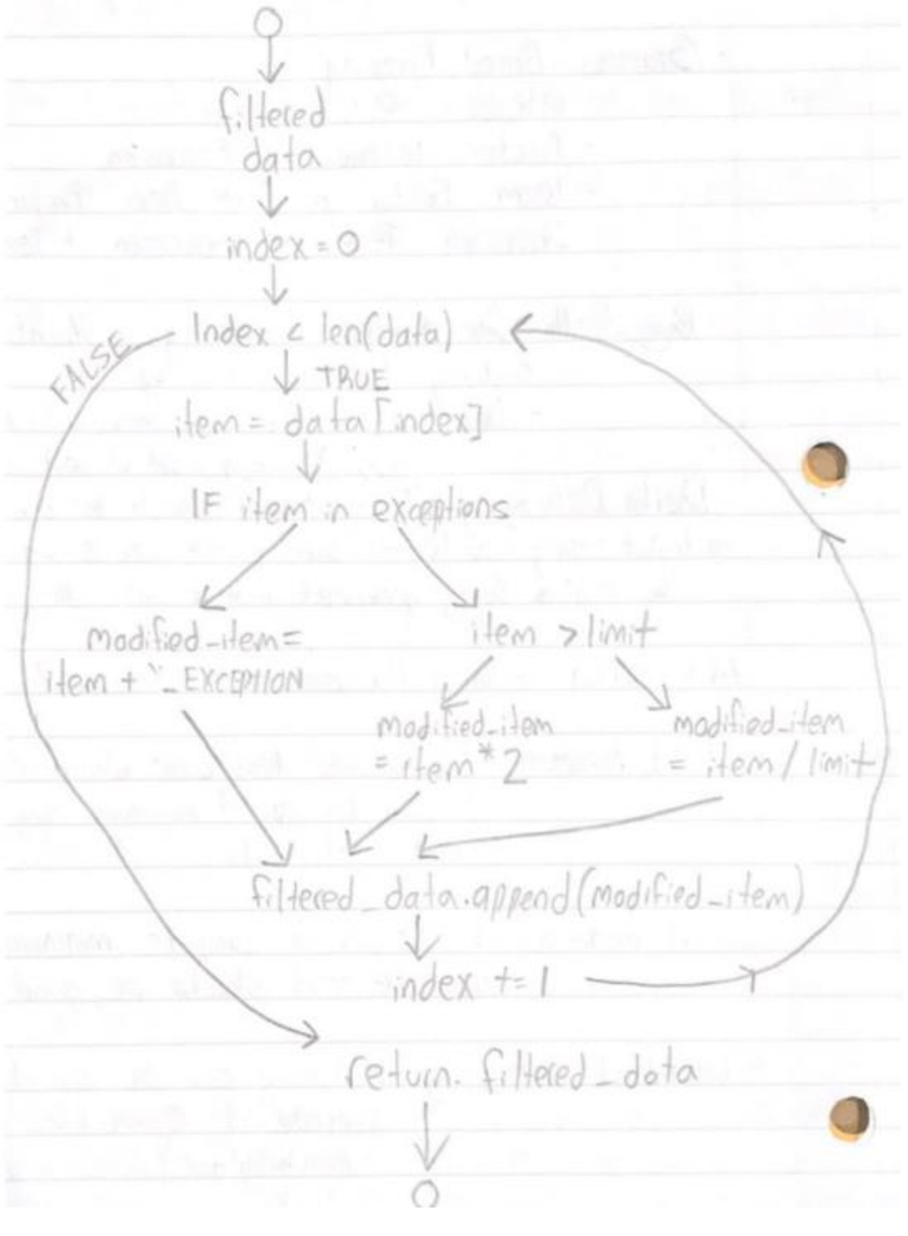
Context-Free Grammar for my test cases:

RandomInput → A random array of size "length" of random Integers

Length → Integer | Integer Integer | …

Integer → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9|

3. A)



3. B) To randomly test this code, I would set the random generation to pick values for Data, Limit, and Exceptions within the constraints of these parameters. I would ensure these random values are a large range of inputs, which also include unorthodox or unpopular inputs such as edge cases, differing sized inputs, etc.

Next, I would use a random test case generator to create a significant number of test cases to run on the code. Passing these random values into the function would return many different outputs as the inputs traversed the pathways in the flow chart. Each test would run the filterData function and provide an output in filtered_data.

These outputs would be recorded and compared with the expected results. After comparing the actual and expected outputs, I could determine if anything went wrong in the calculation and make note of the input that caused this.

Finally, with a list of erroneous inputs, I would be able to recognize failing patterns, leading me to troubleshot and refactor the code to improve the logic to either reject or accept these erroneous inputs to generate acceptable outputs and avoid these failing patterns.

4. A)

- Test Case 1:

Data = [1, 2, 3, 4, 5] **Limit** = 5 **Exceptions** = 1

Code Coverage: 8 out of 9 statements were executed. (88%)

Elif item > limit was not executed due to none of the items being greater than the limit of "5".

- Test Case 2:

Data = [9, 2, 7, 4] **Limit** = 10 **Exceptions** = None

Code Coverage: 7 out of 9 statements were executed. (77%)

If item in Exceptions is not executed, due to there being no exceptions.

Elif item > limit was not executed due to none of the items being greater than the limit of "10".

- Test Case 3:

Data = [8, 12, 25] **Limit** = 10 **Exceptions** = 7, 20

Code Coverage: 5 out of 9 statements were executed (55%)

This input executes modified item = item * 2, modified_item = item / limit, filtered_data.append(modified_item), index +=1 and return filtered_data statements.

- Test Case 4:

Data = [Empty] **Limit** = 10 **Exceptions** = None

Code Coverage: 3 out of 9 statements were executed (33%)

This input executes the filtered_data, index=0 and return filtered_data statements.

B)

```
def filterData (data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + 2
elif item > limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

```
def filterData (data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item + 2
elif item > limit:
modified_item = item / 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

```
def filterData (data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item * 0
elif item > limit:
modified_item = item + limit
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

```
def filterData (data, limit, exceptions):
filtered_data = []
index = 0
while index <= len(data):
item = data[index]
if item in exceptions:
modified_item = item
elif item > limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

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

```
def filterData (data, limit, exceptions):
filtered_data = []
index = 0
while index < len(data):
item = data[index]
if item in exceptions:
modified_item = item
elif item >= limit:
modified_item = item * 2
else:
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data
```

C) The test cases all have varying results on the mutated codes, with a wide range of consistency.

Ranked (most effective to least effective):

Test Case 2: Revealed differing outputs from the majority of the mutated codes.

Test Case 1: Revealed differing outputs from some of the mutated codes.

Test Case 3: Revealed some differing outputs from the mutated codes.

Test Case 4: Not too helpful as it is an empty set and does not provide a wide range of results from the mutated codes.

D)

- **Path Static Analysis**: Ensuring all paths of the program are executed. To use this method on the above code, we would count how many paths on the Flow Chart are covered by the test input.

- **Branch Static Analysis**: To utilize branch static analysis on the above code, we would need to make sure that each possible outcome from each condition is executed at least once. This would require a large number of inputs specifically created to satisfy the conditions in the code.

- **Statement Static Analysis**: We can use this method by checking which statements are executed from the code, such as how code coverage percentage is calculated.

5.

a) The line "*output_str += char * 2*" duplicates the numeric values in the string, which violates the requirement of numeric values to be unchanged. For example, if an input string contained the number "3", in the output it would now be "33", changing the value.

The cause for this bug is the fact that a character is being duplicated. One fix may be to disable altering the output string if it detects a number instead of a character.