Assignment 1

Name: Tristan Arana Charlebois
Student ID: 6680672
Github: https://github.com/git-iso/3P95_assign1

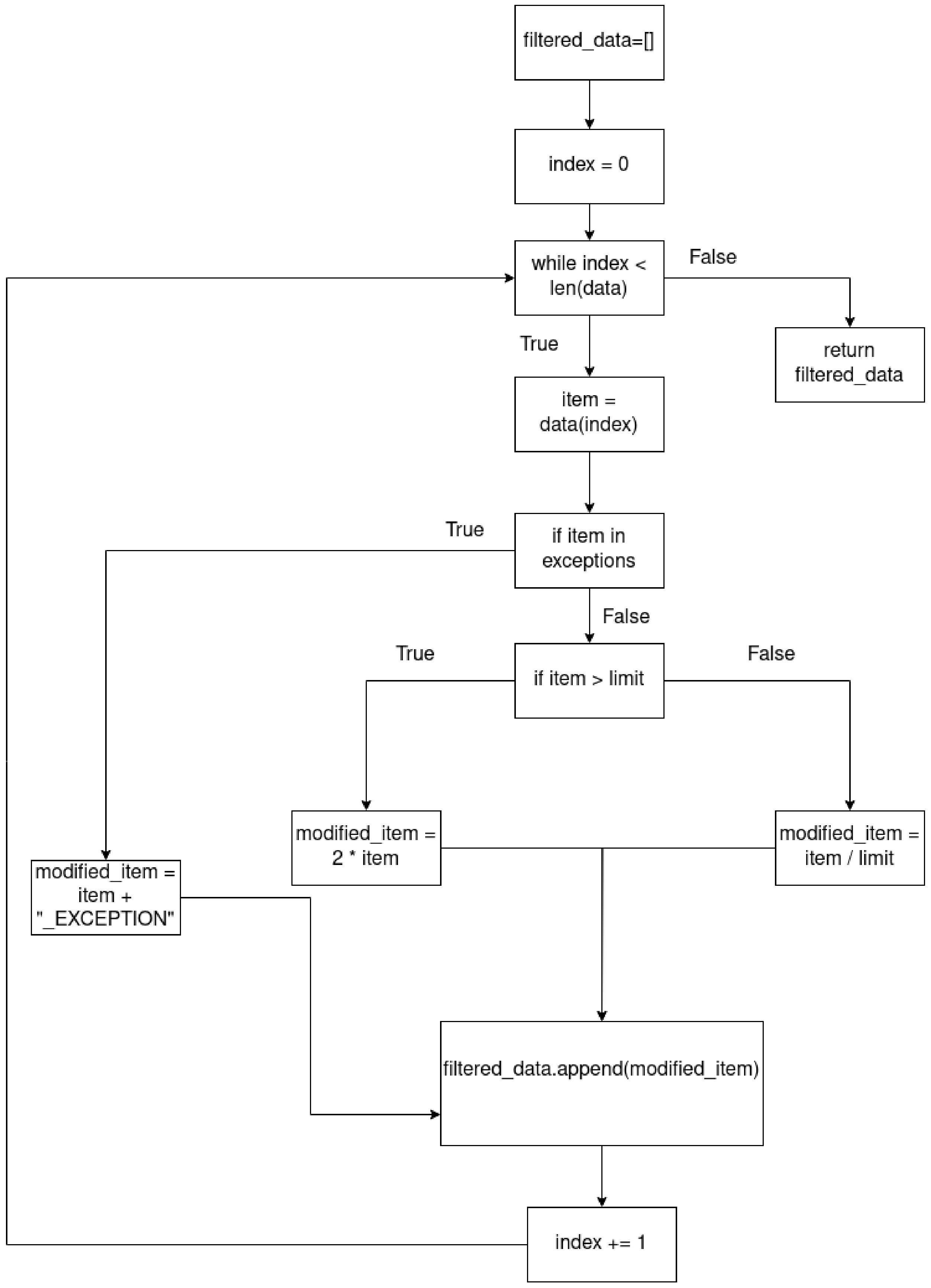
QUESTIONS

1. In software analysis, sound analysis means that, for the bugs that the analysis reports, all of them are true bugs. It never accepts false bugs, but might miss some bugs. A complete analysis means that it reports all of the bugs in the program, but might report false bugs. False positive means that an analysis reports a bug that isn't a real bug, while a false negative does not report a real bug. A true positive means the analysis reports a real bug, while a true negative means that the program does not report a fake bug. All of these definitions are assuming that "positive" means finding a bug. If it instead refers to not finding a bug, then the definitions will flip.

2. a) The algorithm I created is an implementation of bubble short for putting integers in ascending order. I compared it to Java's built-in implementation of array sorting. A number of random integer arrays (default 10) are generated, and the results of sorting are compared between the 2 methods to see if any bugs are generated, which they don't. To see test cases fail, the algorithm can be edited to check if sort[i] < sort[j] instead of >, which generates bugs in some cases.

b) A context-free grammar to generate all possible test-cases:
elements -> element | element "," elements
element -> digits
digits -> digit | digit digits
digit -> "0" | "1" | ... | "9"

3. a)


filtered_data=[]
index = 0
while index < len(data)
False
return filtered_data
True
item = data(index)
if item in exceptions
True
False
if item > limit
True
False
modified_item = 2 * item
modified_item = item / limit
modified_item = item + "_EXCEPTION"
filtered_data.append(modified_item)
index += 1

b) To randomly test this code, I would create a random testing function that would have test cases for exceptions, data (both as an array), and limit variables, with different situations (limit being lower than all data values, exceptions being empty, exceptions being full, etc.). The function would randomly select the test case to use, looping through them multiple times. As the code loops through the data array, it will add each item to the filtered_data array with EXCEPTION or multiplying/dividing by 2. By checking the input values with the expected output values (items in exceptions should say EXCEPTION, items greater than the limit should be greater than what they started out as because of multiplication, etc.) in filtered_data, the random testing function can verify that all possible situations are handled correctly by the code without generating errors or bugs.

4. a) 1. Data = 1, 2
Limit = 3
Exceptions = 1, 2
This test case will only cover the first if statement for "item in exceptions". The code coverage is ~46% (6/13), covering only the initializing statements, first if statement and final statements.

2. Data = 1, 2, 3, 4, 5
Limit = 0
Exceptions = empty
This test case will cover the second if statement when the given item is greater than or less than the limit. The code coverage is ~54% (7/13), covering the initial statements, second else if statement and final statements.

3. Data = 1, 2, 3, 4, 5
Limit = 5
Exceptions = empty
This test case will cover the third if statement when the given item is less than or equal to the limit. The code coverage is ~54% (7/13), covering the initial statements, third else statement and final statements.

4. Data = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Limit = 6
Exceptions = 2, 4, 6, 8, 10
This test case will cover all possible outcomes of items and exceptions. The code coverage is 100% (13/13), covering the entire program.

b) 1. Replace item in exceptions with not in:

```
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. Replace item > limit with <:

```
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. Replace index < len(data) with >:

```
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. Index += 1 becomes = data.length

```
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 = data.length
                    return filtered_data
```

5. Remove filtered_data.append:

```
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

                         index += 1
                    return filtered_data
```

6. Remove final else statements:

```
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

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

c) ORDERED MUTATION SCORES:

```
Mutation score of test case 1 is: 0.6666666666666666
Mutation score of test case 2 is: 0.8333333333333334
Mutation score of test case 3 is: 1.0
Mutation score of test case 4 is: 1.0
```

I got this answer by testing each mutation (6) for all of the test cases (4) and getting the average value (killed mutants / total mutants). A higher number means that more mutants were failed by the test suite, so test case 3 and 4 failed all 6 of the mutants.

d) To evaluate the above code with path, branch and statement static analysis:
For path analysis, by looking at the different execution paths of the program, there is the modified_item = item + "_EXCEPTION" in the first if statement, modified_item = item * 2 in the second if statement, and modified_item = item / limit in the else statement that modify the final answer modified_item that is stored in filtered_data. I would ensure that the program handles these paths correctly on every run, and test every possible control flow path so every combination is verified.
For branch analysis, there are the if item in exceptions branch, else if item > limit branch, and the else branch, along with the while loop these terms are in. I would check each branch's conditions independently to ensure consistency in their output, testing both the true and false results of each.
For statement analysis, every statement, including declaring variables, assigning variables, conditional statements (if, while, etc.), incrementation of index, and adding to filtered_data require review to ensure that they are performing how they appear without generating exceptions.

For all of these analysis types, tools like code analyzer can be used to automate the work. Test cases like the ones previously used with mutations help quantify the effectiveness of the program.

5. a) The bug in this code is on line 7, "output_str += char * 2". The function is supposed to keep numeric characters unchanged, but instead the code multiples the character value by 2. I found this bug by manual review, since it's only a one-line error. To fix it, change the line to "output_str += char".

b) Original Input: abcdefG1
Min Input: 1
Original Input: CCDDEExy
Min Input: CCDDEExy
Original Input: 1234567b
Min Input: 1
Original Input: 8665
Min Input: 8

The delta-debugging algorithm uses a divide-and-conquer strategy to minimize the input values and find the region that generates bugs in the program. A check integer starting from 0 is used to take parts of the input string and check if the proper program generates the same output as the buggy program. If they do, a new minimum is found. It continues looping until it reaches the length of the normal input minus the check value. Check is then incremented by 1 and it continues. By the end, when the check reaches the same length as the whole input length, the last minimum input assigned is the true minimum that generates bugs.
For this program, the primary bug is the multiplication of digits by 2, while numeric characters should actually remain the same as they are. The test cases show that, in every input with numbers, a bug is generated on a single digit (and in the test with only alphabet characters, no bug is found, so the min input is the original).

6. https://github.com/git-iso/3P95_assign1