Link to github repository : https://github.com/nhawryla/3p95assignment_1

1 :

Sound : a tool will be sound if it is able to detect all bugs without reporting any false positives
Complete : will detect all vulnerabilities or bugs however these may be false

True positive : accurately finds a bug within the code
True negative : accurately states that there is no bug
False positive : will show that there is a bug within the code without there being one
False negative : tool does not report a bug that is in the code (overlooked)

These terms can change depending on when the goal is also changed. However, between finding and not finding the bug the only difference would be that the terms would be reversed. For instance if we went from positive meaning we found a bug to not finding a bug, then we would switch the definitions of false positive and false negative we would also switch the definitions of true positive and true negative.

2 :

This example was done using the bubble sort method to sort through an array of random size with randomly selected variables. As far as sorting wise there is no way to break this sorting method when just inputting random variables the only way it could get broken is if a non numerical value is selected to be sorted. The only bug that apparead while running the code was during randomly selecting the size of the array. Randomly selecting this value is problematic because there are multiple ways that doing this will report an error. The two main ways are generating a size so large that the program cannot contain all of that data, and generating a negative number which an array cannot be of size negative. To easily fix these problems, when generating a random number we can specify a certain range. In doing so it will generate a number from 0 to this range ensuring that it will neither be 0 nor too large for the program to handle. To make sure that the test was accurate at the end included is a test to make sure that the array is sorted into ascending order, if it is the program will print true to represent this.

Context free Grammar

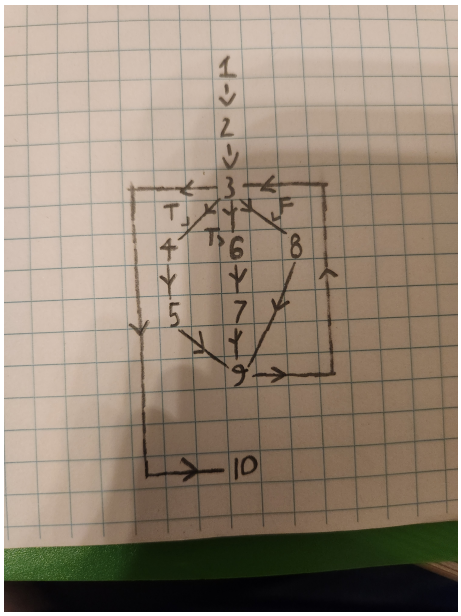
size :== I
I :== “0” | “1” | … | range
range :== “0” | “1” | … | “20”

A[size] (array) :== Term
Term :== I | Term “,” Term
I :== -∞ | ... |∞

3 :

[Number beside lines represents number in flow chart]

```
def filterData(data, limit, exceptions):

filtered_data = [] (1)
index = 0 (1)

while index < len(data): (2)

        item = data[index] (3)

        if item in exceptions: (4)
                modified_item = item + "_EXCEPTION (5)
        elif item > limit: (6)
                modified_item = item * 2 (7)
        else:
                modified_item = item / limit (8)

        filtered_data.append(modified_item) (9)
        index += 1 (9)

return filtered_data (10)
```



B)

To Random test with the given code the best way to do it would be to randomize the data passed into the method for instance data, limit, and exceptions. In randomizing these parts of data we will be able to determine what is able to break this system. With random testing some combinations of data could be proven to cause errors in the code. If we repeatedly randomize all of the values we may eventually be left with a set of combinations that will not work for the code. The only other piece of data that could be randomized is the index however this could very easily break the code and probably should not be done.

4 :

A)

Test cases

1. When the length of data is equal to 0 meaning there is no data stored the code coverage is 30%. This is because only 4 lines out of 13 are actually being run through which are the lines up to and including the while check and then since it won't be true we jump to the return statement.

   ([],4,1) input

2. Next would be when exceptions holds all possible values contained in data meaning that every item in data is an exception this would only cover 9 lines of code, because we would ignore all other if branches and jump right to the next part in the code which would leave us with code coverage of 9/13 which is 69% code coverage

   ([1,1,1],4,1) input

3. After that would be when some items contained in data are exceptions and are greater than the given limit. This would result in 11 lines of code being cover which includes the next line of if statements which would cover 84% of code

   ([1,5,6],3,1) input

4. Lastly would be code coverage of 100% when the data contains elements that fulfill all if and else statements in the code so it would include exceptions, items > limit, and items that do not meat either of these standards.

   ([1,2,3,5,6],4,1) input

B)

Mutations :

1. Elif item > limit (changed to) elif item < limit
2. If item in exceptions (change) if item not in exceptions
3. While index < len(data) (change) while index >= len(data)
4. Index += 1 (change) index += 2
5. modified _item = item * 2 (change) modified _item = item / 2
6. modified _item = item / limit (change) modified _item = item * limit

C) numbers in the test correspond to numbered mutations above

Test 1 ([],4,1) : 4th best only finds 1 mutation

1. Does not find mutation
2. Does not find mutation
3. Will find mutation (while loop will run once)
4. Does not find mutation
5. Does not find mutation
6. Does not find mutation

Test 2 ([1,1,1],4,1) : 3rd best misses 2 mutations

1. Will find mutation (1 is less than 4)
2. Will find mutation (1 will not be listed as exception)
3. Will find mutation
4. Will find mutation (will skip over numbers)
5. Does not find mutation
6. Does not find mutation

Test 3 ([1,5,6],3,1) : 2nd best only misses 1 mutation

1. Will find mutation
2. Will find mutation
3. Will find mutation
4. Will find mutation
5. Will find mutation
6. Does not find mutation

Test 4 ([1,2,3,5,6],4,1) : Best finds all mutations

1. Will find mutation
2. Will find mutation
3. Will find mutation
4. Will find mutation
5. Will find mutation
6. Will find mutation

D)

Branch : To test the code using this method I would first lay out all of the possible branches then I would throw some test cases through to see what branches are actually being run through and getting triggered by certain inputs.

Path : With this method I would make a flowchart that will depict all possible paths the code is able to take. I would then create a list of all possible paths the code is able to take and then run some test cases to make sure all paths in the code are being run through properly, and also make sure that all paths have a possibility of being triggered and run through.

Statement : For this method I would walk through the code statement by statement with multiple test cases, and make sure all statements have a possibility to be covered by any of the tests. Sometimes it can be easy to add statements that won't be covered by any test cases meaning that they are most likely useless and unneeded for the code.

5 :

A)

The error in this code is on line 7 and it happens when a char is being multiplied by an integer. This is technically allowed in most languages. A char is stored as a numerical value however this can be easily mixed up because in the instance of char c = ‘8’, it is not stored as the number 8 but actually is 56 therefore when you multiply c by 2 u dont get 16 but instead get 112. The strategy I used to find this bug was by first looking at the code to see if anything stood out to me and then once I pinpointed that line 7 looked funny I created a simple test case which proved that this line was in fact the bug.

B)

For delta debugging to work in this situation I needed to figure out a way to check if the output is correct or not. This would be a little more difficult because when the output is wrong it doesn't actually report an error message, so to fix this problem I slightly changed the original code to now return true or false instead of the usual output (true meaning a bug was found false meaning no bug was found). What might be the bug however, well we already know why the program doesn't work correctly which is multiplying the char number by 2, using this knowledge I changed the code again to add a check when a numerical char is found. This check sees if the actual numerical value of the char multiplied by 2 and the normal chars value multiplied by 2 are equal to the same, which we already know will not. When this happens the method will return true meaning that it has found the error in the code. Here is a visible representation char(1) * 2 =

49 *2 = 98 compared with int(1) * 2 = 2. Now onto the actual delta debugging for this I created a method which takes in an input of a string. Next it checks whether the input contains an error if so it splits the input into left and right and checks both sides. It will continuously break down the string until it is left with the exact error in the code. If time was a complication and we needed to reduce more time and not get exact answers instead of letting the code go all the way to the base, we can tell it to stop at a certain input length leaving us with a close enough approximation guess of where the error lies.