Question 1

Sound Analysis: Soundness is the ability to not report false positives. This means that it will never report a bug that does not exist. Sound analysis prevents false positives. Complete Analysis: Completeness is the ability to report all bugs and vulnerabilities. This means that if there is a bug, it will be reported. Complete analysis prevents false negatives. These are different, as sound analysis will only report bugs that do exist, and complete analysis will report all bugs, even if they are a false positive.

True Positive is if a bug exists, it is reported. True Negative is if there are no bugs, nothing is reported. False Positive is when a bug doesn't exist but is flagged as a bug. False Negative is when a bug exists, but the system does not report it as a bug.

The terms would swap when positive is finding a bug compared to when positive is not finding a bug. The analysis is positive when a bug is not found, true positive would mean that there were no bugs found, and true negative would mean that bugs were found. False positive would then mean that there was a bug, but the system didn't report it, and false negative would mean that the system found a bug that doesn't exist.

Question 2

2A)

A)

Question2.java attached

B)

I took a generic sorting algorithm from GeeksforGeeks https://www.geeksforgeeks.org/sorting-in-java/. I replaced the given array with an array parameter for getting the random test cases. I created my test case generator using java.util.random. I created the max size of the array to be 20, as I didn't want the output to be too long, it will then go through a for loop generating numbers with a max value of 100, the test case method will then return the generated array. In the Main method, an array will be created using the array generated by the test case method, that array will be used by the constructor where the sorting algorithm method is called, it will then sort the randomly generated array. A separate method is then called after to print the array.

D)

All the code is in one file, if you are using an IDE (IntelliJ, VSCode, etc), compiling and running it will work. Using terminals, just running the Sort.java file will run the entire code. The test case only prints once, so it will need to be ran multiple times to get different outputs.

E)

Test 1:

Unsorted array:

10 22 13 35

Sorted array:

10 13 22 35

Test 2:

Unsorted array:

34 45 82 89 57 61 100 28 92 64 95 73 60 86 93

Sorted array:

28 34 45 57 60 61 64 73 82 86 89 92 93 95 100

Test 3:

Unsorted array:

6 89 64 63 36 84

Sorted array:

6 36 63 64 84 89

2B)

Test Cases -> Action

- Action -> Generate “random array”
- Action -> Print “random array”

Random array -> sorting algorithm

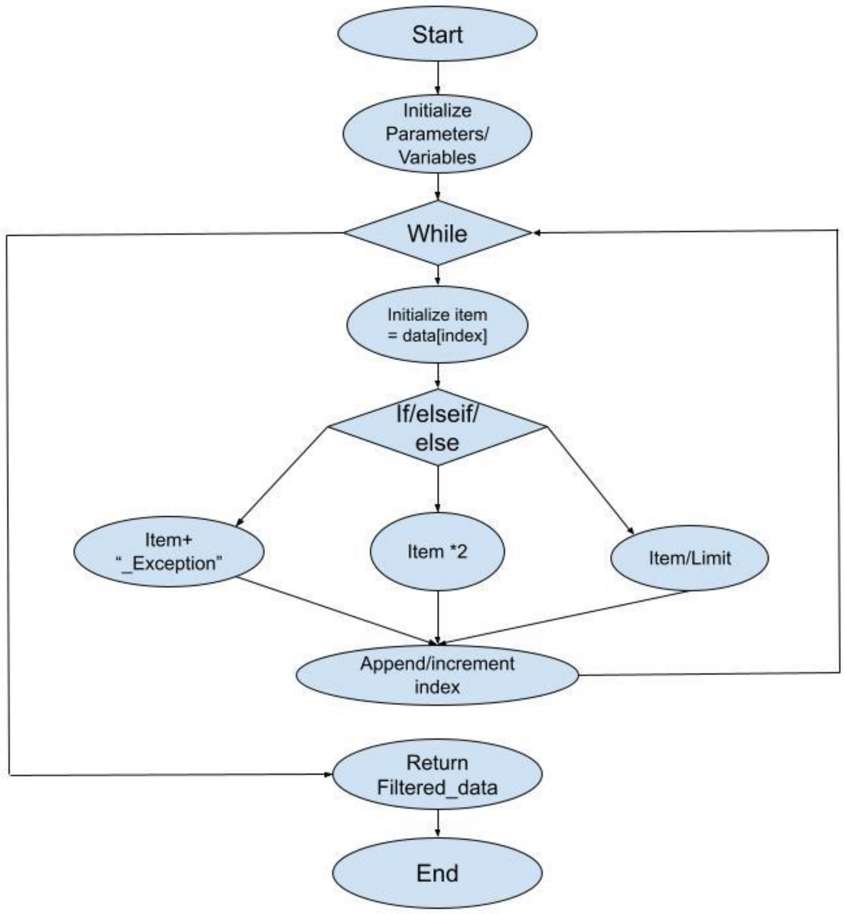
Sorting algorithm -> Action

- Action -> generate “sorted array”

Sorted array -> Action

- Action -> Print sorted array

Question 3



3B)

To do random testing on this code, I would generate random input of data for each of the parameters (data, limit, exceptions). Data would consist of randomly sized arrays of data, limit would randomly generate an integer value to be the limit, and exceptions would randomly generate exceptions. These parameters would then be used to fill the filtered_data array and run through the while loop.

Question 4

4A)

Test Case 1

Data = [3], limit = 4, exceptions = [4,5], code coverage = 30%

Test Case 2

Data = [1,2,4,5], limit 4, exceptions = [2,5], code coverage = 60%

Test Case 3

Data = [1,2,3,4,6], limit = 3, exceptions = [1,3,4], code coverage = 90%

Test Case 4

Data = [1,2,3,4,5,6], limit = 3, exceptions = [1], code coverage = 100%

4B)

Mutation 1

index = 0 -> index = 1

Mutation 2

If item in exceptions -> if item not in exceptions

Mutation 3

Elif item > limit -> elif item < limit

Mutation 4

Elif item > limit -> elif item == limit

Mutation 5

modified_item = item * 2 -> modified_item = item * 3

Mutation 6

index += 1 -> index += 2

4C)

1. Case 4 (detects all 6 mutations)
2. Case 3 (detects 5 mutations (1,2,3,4,6))
3. Case 2 (detects 4 mutations (1,2,4,6))
4. Case 1 (Detects 2 mutation (1,6))

4D)

Path Coverage: Path coverage analyzes every route that could be taken and generates a test case for that route. You can use this for the code above by checking which routes are taken, for example: if the data had a value of 1, and the limit was 2, and had no exceptions, since that value was less than the limit, it would fall under the else statement. You can examine each test case individually this way to see if your code is missing anything.

Branch Coverage: Branch Coverage makes sure that each branch is taken at least once. You can use it to see where the code went using if-else statements, you can see which branches have and haven't been taken using this method. Using the same example from the path coverage, the branch taken would've been the else branch.

Statement Coverage: Statement Coverage ensures that each statement has been used at least once, this will analyze each choice from the branches, and show which statements have been used and which ones haven't. Again, using the example from the path coverage, it would show that the else statement was used, but the if and the else if statements were not.

Question 5

5A)

The question states that if the character is a numeric value, the value is supposed to remain unchanged, however, elif statement alters the output string by multiplying the value by 2. I used static analysis by looking through the code and relating it to the question (elif was supposed to be unchanged, but wasnt).

Question 6

https://github.com/dc19kg/COSC3p95