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.***

   An analysis tool's "soundness" refers to it being able to avoid generating false positives for bug testing, while its "completeness" refers to the tool's ability to detect all real vulnerabilities (true positives) in the software.

   A "true positive" is when an analysis tool detects a real bug in the software, but a "false positive" is when the tool flags a bug in the software that does not exist. A "true negative" is when the tool does not detect a bug in the software where there is none, but a "false negative" is when the tool does not detect a bug when there is one in the software.

   That is how those terms would be defined if "positive" meant finding a bug, but if it meant not finding a bug instead, the definitions for true positive and true negative would be switched. Similarly, the definitions for false positive and false negative would be switched as well. The definition for "sound" would then refer to the tool being able to avoid generating false positives (in the new way it is defined) and "completeness" refers to generating only true positives (in the new way it is defined).

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.***

   a. (See code at the end of the document)

      The bug for my sorting algorithm is the algorithm ignoring values of 3 during its sort.

      My sorting algorithm takes a list of integers and identifies the positions in the said list that hold the value 3. Those positions are saved to another list and the remaining values in the list are stored in a new "leftover" list. This leftover list is then sorted using Python's built-in list sorting algorithm [list.sort()], and a new list is generated where the positions that contained 3 in the original list are 3 again, but the remaining positions are occupied by the numbers from the newly sorted leftover list. The algorithm then returns the resulting list.

      The code can be run by opening the folder with the code files in visual studio code and executing the testSortingAlgo.py file.

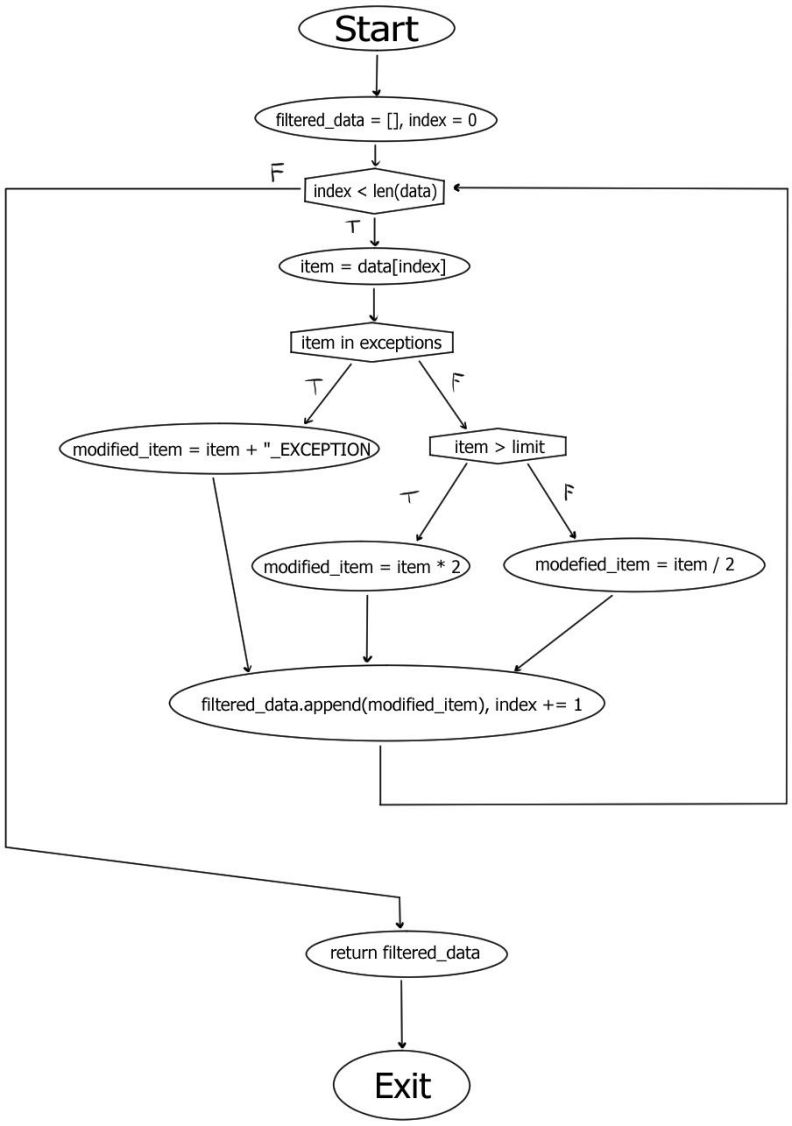
b. ***Provide a context-free grammar to generate all the possible test-cases.***

test_case ⟶ “[” Elements “]”
Elements ⟶ Term | Term “,” Elements
Term ⟶ "1" | "2" |"3" |"4" |"5" |"6" |"7" |"8" |"9" |"10"

3.

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

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.



(The code said “modified_item = item/limit”, but I assumed it meant “modified_item = item/2” due to the question specification)

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.***

To randomly test the code, I would create a secondary method or program that randomly generates data arrays of varying lengths, a randomly generated limit, and a randomly generated list of exceptions. The program/method would then input these randomly generated parameters into the filterData method and verify the returned filtered_data.

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***

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

Case 2:
data = [2, 3, 2, 3], limit = 3, exceptions = [1, 3], code coverage = 90%

Case 3:
data = [], limit = 4, exceptions = [], code coverage = 30%

Case 4:
data = [1, 2, 3], limit = 2, exceptions = [1, 2, 3], code coverage = 70%

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

Mutation 1: Changing “elif item > limit” to “elif item < limit”
Mutation 2: Changing “if item in exceptions” to “if item not in exceptions”
Mutation 3: Changing “modified_item = item * 2” to “modified_item = 0”
Mutation 4: Changing “return filtered_data” to “return data”
Mutation 5: Changing “elif item > limit” to “elif item == limit”
Mutation 6: Changing “index += 1” to “index += 2”

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.***

The ranking for the test-cases are as follows:

i. Case 1 (detected all 6 mutations)
ii. Case 2 (detected 4 mutations [mutations 1, 2, 4, 6])
iii. Case 4 (detected 2 mutations [mutations 2, 4])
iv. Case 3 (detected 1 mutation [mutation 4])

d. ***Discuss how you would use path, branch, and statement static analysis to evaluate/analyze the above code.***

I would choose a test case, like case 1 described earlier, which would provide complete coverage over every path, branch, and statement. I will then use this test case to determine if the program gives the desired result.

5.

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

Bug 1: Within the elif statement, the numeric character undergoes string repetition by 2 instead of being left alone

I found this by manually reviewing the code.

b. (See code at the end of the document)

My delta-debugging algorithm splits recursively divides a given string and feeds it to the "prossesString" method until it finds the characters that are causing the error. It checks to see if the length of the string increases after being processed, because this will indicate that string repetition has occurred due to the bug in the code.

6. https://github.com/Lanoswego/COSC-3P95-A1

Question 2a Code:

[sortingAlgo.py]

```
class sortingAlgo:

    """This class sorts a given list using on of two methods. The method used is
determined by the value passed"""



    #This method sorts the list of integers excluding positions with 3 ("bug"
that ignores 3's in the list)

    def altSort(self, list):

        listSize = len(list)        #size of the original list

        indicies = []               #list of positions in the list that are 3

        leftover = []               #list of items from the list that are not 3

        result = []                 #resulting "sorted" list of integers



        #checks the pairs of [index, int] for the list and adds the index number
to the indicies list if int = 3. Otherwise it adds the int to the leftovers list
```

```
        for pair in enumerate(list):
            if pair[1] == 3:
                indicies.append(pair[0])
            else:
                leftover.append(pair[1])

        if len(indicies) == 0: return sorted(leftover)      #if there are no 3's
in the list, simply return the sorted list

        leftover.sort()     #sorts the leftover integers from the list

        currentPos = 0      #current position in the results list
        indPos = 0;         #current position in the indicies list

        #generates a new list by placing the value in sorted order around the
positions of any 3 in the original list
        while currentPos < listSize:

            #if this position had a 3 in the original list, then place three for
that position in the result list
            #else take the int at the start of the leftover list and place at the
current position in the result list
            if currentPos == indicies[indPos]:
                result.append(3)                                        #adds 3
to the result list
                if indPos < len(indicies)-1 : indPos = indPos + 1
#increases indicies position
            else:
```

```
                result.append(leftover.pop(0))              #takes int from the
beginning of the leftover list and adds it to the result list


            currentPos = currentPos + 1                     #increases result
list position


        return result
```

[testSortingAlgo.py]

```
import random

import sortingAlgo as algo


class testSortingAlgo:


    def __init__(self):

        self.intList = []

        self.sortingAlgo = algo.sortingAlgo()

        return


    #This method generates a list of integers (between 1 and 10) where the
elements of the list are randomly generated. Range of the list size is 1 to 100.

    def randomList(self):

        listSize = random.randint(1,20)     #generates a random integer between 1
and 20 for the list size


        while len(self.intList) < listSize:

            self.intList.append(random.randint(1,10))   #generates a random
number between 1 and 10, and adds it to the array
```

```
        return


    #generates a string form of the list given
    def generateStringList(self, list):
        l = "["

        for num in enumerate(list):
            if num[0] == len(list) -1:
                l += f"{num[1]}"
            else:
                l += f"{num[1]}, "
        l += "]"


        return l




    #takes a randomly generated list, sorts it with the sorting algorithm, and
checks the sorted list for errors
    def testSort(self):
        self.randomList()
#generates a random list of integers

        print(f"Original list: {self.intList}")
#prints the original list

        self.intList = self.sortingAlgo.altSort(self.intList)                #alt
sorts the list
```

```
        print(f"Sorted list: {self.intList}")
#prints the original list


        #if the list was sorted without errors print positive result, otherwise
print negative result
        if self.intList == sorted(self.intList):
            print("Result: List was sorted CORRECTLY")
        else:
            print("\nResult: List was sorted INCORRECTLY")



test = testSortingAlgo()
test.testSort()
```

Question 5b Code

[testprocessString.py]

```
#this class finds the set of characters within a string that result in an
incorrect output
class testProcessString:


    def __init__(self):
        self.errorSet = []
```

```
    #converts a string's uppercase values into lowercase values and vice-versa
    def processString(self, 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


    #uses binary search on an input to determine the characters in the string that cause the errors
    def deltaDebug(self, str, i):
        #if the sting is empty just return
        if len(str) == 0:
            return
        #elif the string has one character and causes an error, add that character to the error set if it is not already in it
        elif len(str) == 1 and len(str) != len(self.processString(str)):
            if str not in self.errorSet: self.errorSet.append(str)
            return
        #elif the string generates an error
        elif len(str) != len(self.processString(str)):
            self.deltaDebug(str[:(len(str)) // 2], i+1)   #checks the left half of the string
            self.deltaDebug(str[(len(str)) // 2:], i+1)   #checks the right half of the string
```

```
        #if this is the end of the root level, print error set and clear error
set
        if i == 0:
            print(f"Error Set for {str}: {self.errorSet} \n")
            self.errorSet = []

        return



test = testProcessString()          #creates instance of testProcessString class


#tests various inputs to find the error sets within them
test.deltaDebug("abcdefG1", 0)
test.deltaDebug("CCDDEExy", 0)
test.deltaDebug("1234567b", 0)
test.deltaDebug("8665", 0)
```