COSC 3P95 – Assignment 1
Wednesday October 11, 2023
Andrew Pauls

Question 1

Two qualities that testers look for in the analysis of software is that of soundness and completeness.

An analysis is sound if bugs identified by the testing cases are all real bugs. An analysis that is sound therefore does not report any false positives. An analysis that is complete will have its test cases uncover at least all of the existing bugs within the software. In complete analysis, every bug that exists will be identified by the testing suite. It will likely be the case that “bugs” that don’t actually exist will be detected in addition to the ones that are real. A complete analysis therefore does not produce any false negatives.

In summary, the difference between a sound and complete analysis is that a sound analysis does not report any false positives, and a complete analysis does not report any false negatives.

Below are the definitions for the four categories of bug identification.
1 – True Positive
When during analysis, test cases identify that a bug exists, and that bug actually exists.

2 – True Negative: When during analysis, test cases do not detect any bugs for a segment of code, when in fact no bugs exist for that portion of code.

3 – False Positive: When during the analysis of software, the testing program indicates that a bug exists for a portion of code, when in actuality, no such bug exists.

4 – False Negative: When during testing, the testing software do not detect any bugs for a segment of code, when in actuality, some bug exists for that portion of code.

Lastly, how would these terms change if the goal of the analysis changed?

The goal of analysis as we currently understand it is for testing to discover the presence of bugs. Thus, we have been proceeding under the assumption that the identification of a bug is considered to be a ‘positive’ event, and that this event can be true or false, depending on whether the identification was accurate.

If the goal changed so that instead of looking to identify bugs, we were looking to identify portions of code that do not have any bugs, the definitions 1-4 defined above would be shuffled. A true positive would become what was considered to be a true negative. A false negative would become what was a true positive. A false positive would become what was a false negative, and a false negative would become what was a false positive.

Question 2:
a) – MY SOURCE CODE. There are two classes here. One is called main, the other is called Question2. Main comes first.

```
/** main is responsible for creating an instance of Question2, and then running Question2's primary
 * method called 'displayMyWork()'
 *
 * while evaluating my work, main will be the class that is run after compilation.
 */
public class Main {
    public static void main(String[] args) {
        Question2 q2 = new Question2();
        q2.displayMyWork();
    }
}
```

```
import java.util.*;

/**Question2 is a class that holds the data and methods necessary to answer question2 of assignment 1
 * for COSC 3P95, Fall 2023
 *
 * The single variable of this class is a list of integers.
 *
 * There are a variety of methods which generate a random list of integers, display messages to the
user,
 *  and sort the integers. The primary method which demonstrates the capabilities of this class is called:
 *      'displayMyWork()', which takes the user through a comprehensive walkthrough of question2.
 *
 *  Author: Pauls, A.
 *  Date: Oct 10, 2023
 *  ID: 6368914
 *
 */
public class Question2 {
    private static List<Integer> integerArray;

    /** displayMyWork() is called by the main class, and shows off the features of this class.
     * this method is designed to make life easy for markers.
     *
     * the user is greeted on SYSOUT, shown a randomly generated array of integers, shown the array
sorted, and provided
     * with an exit message.
     */
    public void displayMyWork(){
        introductoryMessage();              // welcome the user on SYSOUT
        generateRandomArray();              // produce an array of integers (has randomized length, has
randomized elements)
        printArray(integerArray);           // display the newly developed random array
        messageToUser();                    // update user that the array is now ready to be sorted
        sortArray(integerArray);            // sort the array
        printArray(integerArray);           // display the sorted version of the array
        exitMessage();                      // inform user that the program is complete
    }

    /** exitMessage() is a simple method that talks to the user on SYSOUT
     *
     */
    private static void exitMessage(){
        System.out.println();
        System.out.println("-------------------------");
        System.out.println("That is the end of this program!");
        System.out.println("Thanks, bye");
    }
    /** messageToUser() is a simple method that tells the random array is about to be sorted.*/
    private static void messageToUser(){
        System.out.println();
        System.out.println("-------------------------");
        System.out.println("We will not sort the array and print it to screen for comparison");
    }
    private static void sortArray(List<Integer> randomInput){
        Collections.sort(randomInput);          // sort the array
    }
    /** a simple method to greet the user on SYSOUT */
    private static void introductoryMessage(){
        System.out.println("This is the implementation of question 2, Assignment 1 for COSC 3P95, Fall 
2023.");
        System.out.println("In this implementation, there exists \n - a method which sorts an array of 
integers from lowest to highest" +
                "       \n - a method which generates random integer arrays \n - an automated program which 
shows how these two methods perform on SYSOUT");
    }
    /** print the array passed as parameter */
    private static void printArray(List<Integer> theList){
        System.out.print("[ ");
        for(int i = 0 ; i <theList.size()  ; i++){
            System.out.print(theList.get(i) + " ");
        }
        System.out.print("]");
    }

    /** generateRandomArray() produces a new array each time it is called.
     *  method generates a random integer which will be the length of the array, and
     *  generates a random integer for each cell in the array
     *
     *  there are bounds on the minimum and maximum size that the array can be
     *  there are also bounds on the minimum and maximum value that each element may take
     *  these bounds are arbitrary and can be adjusted easily
     */
    private static void generateRandomArray(){
        integerArray=new ArrayList<Integer>();          // initialize list
        System.out.println();
        System.out.println("An array of random length and of random integer values will now be
generated.");
        System.out.println("Currently generating the random length.  Constraints: 1 <= length <= 50 ");
        int randomlyGeneratedNumberOfNumbers = (int) (Math.random() * ((50) + 1)) +1;          // 0 <=
length <= 50
        System.out.println("We will now fill the array of length: " +
randomlyGeneratedNumberOfNumbers + " with randomly generated numbers between 1 and 25");
        for (int i = 0 ; i < randomlyGeneratedNumberOfNumbers; i++){        // for each of the elements
            int randomInteger = (int) (Math.random() * ((25) + 1));        // generate a random int such that 0
<= number <= 25
            integerArray.add(randomInteger);                              // insert the integer into the array
        }
        System.out.println("The random input array of integers has been generated. Returning input array
now.");
    }
}
```

---

b) There are essentially just two core methods to this class that are specifically asked for in the assignment outline. These two methods must produce randomized integer arrays, and sort integer arrays.

The method generateRandomArray() uses an arrayList to store an arbitrary number of integers. The array produced by this method is of randomized length and composed of random integers. The randomized length is generated by using the Math module’s random() method. I selected the arbitrary lower and upper bounds of 0 and 50. Once the length has been determined, I used the same random() method to generate a random integer value between the arbitrary bounds of 0 and 25. A simple for loop populates the array with the random elements.

The sorting method sortArray() is a single line of code which uses an inbuilt Java method to sort integer arrays from lowest to highest.

The sorting method I selected has not produced any errors thus far during my own testing. As this method is built in, it is highly trustworthy. My array generating method does not have the ability to find any potential bugs in the sorting algorithm for I have given it strict upper and lower bounds on both length and element possibilities. I have also restricted it to generate nothing but integers. As the bounds are small, I will never run out of memory. As the type of values generated are always integers, I will never have a type error. The simple for loop used for populating the random array and to print the array is incapable of going out of bounds.

c) Comments are written in the source code.

d) To compile and run the code:
Produce a new Java Project.
When the default main() class appears, delete the entire provided text.
Copy and paste the main() source code from above (part a).
Then create a new class – name it Question2.
Copy and paste the class file Question2 which is located above.
Complile the project.
Run the main() class.

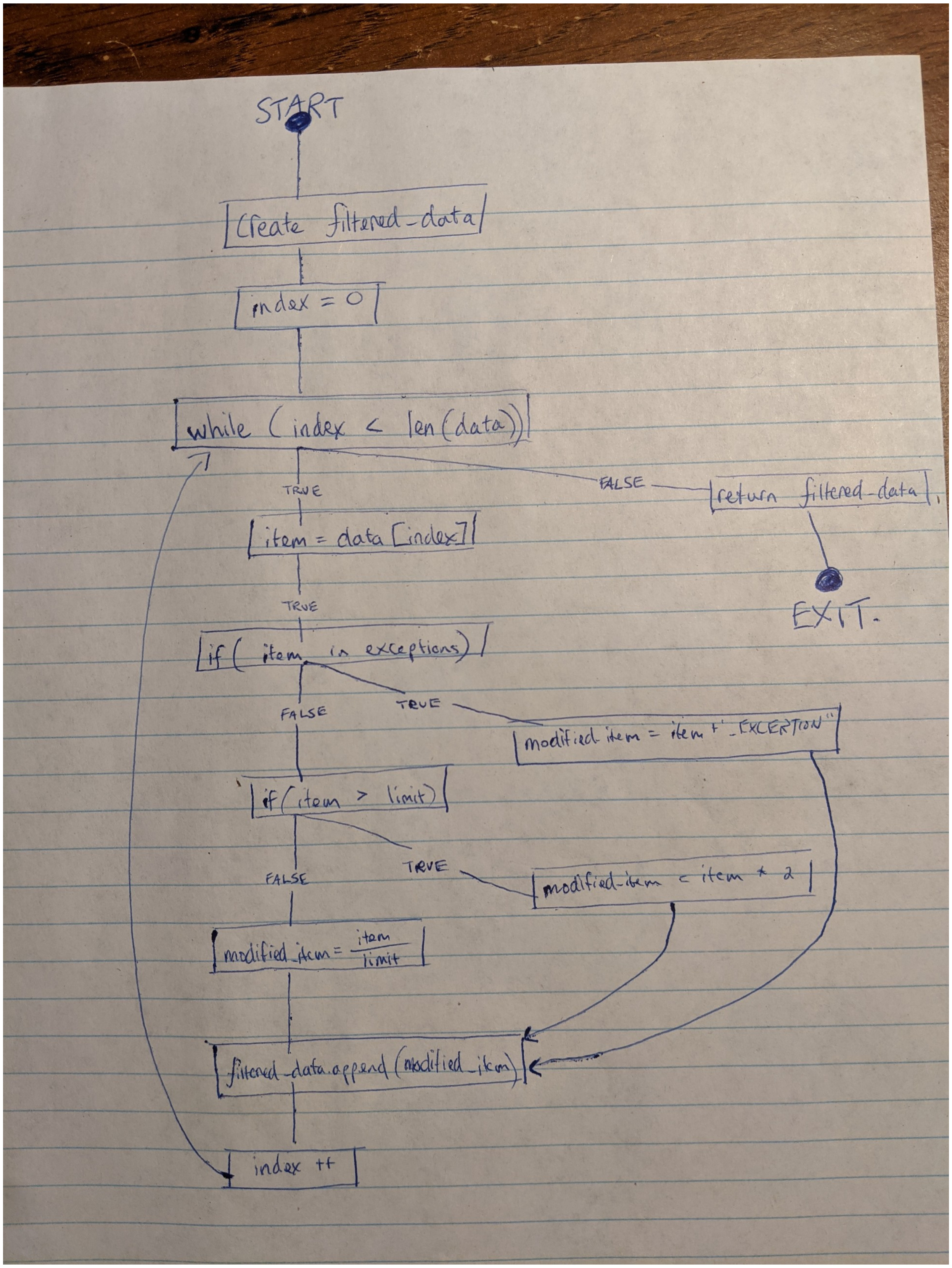
e) Logs are provided to SYSOUT throughout the program.

f) Logs are provided during random test execution, indicating success in every instance.

B) The context free grammar that produces every possible array of integers must have at its smallest value an array of length 0 and at its largest value an array of length 50. Each value may take on the integer values between 0 and 25 inclusive. Thus the CFG is:

Start → Empty String | IntegerString
IntegerString → IntegerString Value | Value
Value → 0 <= 25

Question 3:
a)


START
Create filtered_data
index = 0
while (index < len(data))
TRUE
FALSE
return filtered_data
EXIT.
item = data [index]
TRUE
if (item in exceptions)
FALSE
TRUE
modified item = item + "_EXCEPTION"
if (item > limit)
FALSE
TRUE
modified_item = item * 2
modified_item = item / limit
filtered_data.append (modified_item)
index ++

b) If I were to randomly test the provided code, I would perform a series of deliberate random tests to try and uncover bugs at all levels of the system.

At the most basic level, I would conduct randomized unit testing. All methods and loops would be tested with a relatively large enough suite of random inputs to try and break the methods and loops. I would also make sure that the correct exception handling procedures were in place for potentially faulty input (ex. Input of the wrong type). Testing for incorrect inputs would be performed by generating randomized arguments for the method call.

I would also work deliberately on Path testing. The control flow diagram indicates there are a number of different paths that this program can take. With a large enough and random enough test suite, it would not be infeasible to explore all possible paths. In the event that certain types of random inputs generate bugs more often, I could use genetic algorithms to mutate problematic random inputs slightly, which would have a strong chance of exposing the bug. To ensure all paths are taken, it would be necessary to generate inputs that contain exceptions, items greater than the limit, and items that are neither of the previous two.

To generate sets of randomized inputs for this program, a grammar based fuzzing approach would be effective. It would not be overly difficult to write a C.F.G. that can compute all of the possible inputs for this program.

Question 4
a) – Before I present my four distinct test cases which range from 30% to 100% code coverage, I would just like to assert that from my own analysis of the program, the total number of branch statements is 6 and the total number of statements is 9, plus 3 condition statements governing conditionals and the while loop. This would imply that all 6 branch’s would need to be traversed to have complete branch coverage, and all 12 statements would need to be executed to satisfy complete statement coverage.

Test Case #1:
data = {4, 3}, limit = 5, exceptions = {3, 4}
Code Coverage:

Branch Coverage:	3 / 6	= 50 %
Statement Coverage:	9/12	= 75 %

Test Case #2:
data = {4,3}, limit = 2, exceptions = {4}
Code Coverage:

Branch Coverage:	5 / 6	= 83%
Statement Coverage:	11 / 12	= 92 %

Test Case #3:
data = {4,2}, limit = 5, exceptions = {}
Code Coverage:

Branch Coverage:	4 / 6	= 66%
Statement Coverage:	10 / 12	= 83%

Test Case #4:
data = {1,6,10, 11}, limit = 7, exceptions = {11}
Code Coverage:

Branch Coverage:	6 / 6	= 100%
Statement Coverage:	12 / 12	= 100%

b) – The six mutations

1:	while (index < len(data))	mutated to while (index <= len(data))
2:	item = item[data]	mutated to item = item[data +1]
3:	elif item > lim	mutated to elif item >= limit
4.	elif item > lim	mutated to elif item < limit
5.	modified_item = item / limit	mutated to modified_item = item / 3
6.	index++	mutated to index = index +2

c)
Test Case 1 – Effectiveness
Test Case 1 kills mutants: [1, 2, 6]
Therefore it has a Mutation Score of 3/6 = 50%

Test Case 2 – Effectiveness
Test Case 2 kills mutants [1,2,4,5,6]
Therefore it has a Mutation score of 5/6 = 83%

Test Case 3 – Effectiveness
Test Case 3 kills mutants [1,2,4,5,6]
Therefore it has a Mutation Score of 5/6 = 83%

Test Case 4 – Effectiveness
Test Case 4 kills mutants [1, 2, 4, 5, 6]
Therefore it has a Mutation score of 5/6 = 83%

From worst to best:
Case 1, Cases(2-4)
All four of my test cases reach the inner components of the while loop.
This is why all four test cases kill mutants 1,2,6 minimally. As Case 1 only contains exceptions, it does not detect mutants in the deeper branches (mutants 3,4,5).

Test cases 2-4 are surprisingly of the same mutation score. These test cases were designed to be more than just exceptions. Not one of these test cases contained items in data that equal the limit, and as a result, not one kills the 3rd mutant. As it turns out, test cases 2-4 kill all other mutants, and are therefore of equivalent ability with regard to mutation analysis.

d)
Path Static Analysis: The aim here is to ensure that as diverse a set of possible paths are traversed, so that no whole section of the program is not tested. Ideally, every single path is tested. Some programs are too large to feasibly explore every path. In this example, the number of possible paths is not large, and an exhaustive path analysis could be conducted in a reasonable amount of time. If I had the time, I would start with the shortest paths, and reverse engineer what types of inputs lead to the program taking those paths. I would then move deeper and deeper into the program, reverse engineering what inputs would take which paths.

Branch Static Analysis: Branch Static Analysis would technically be fully covered if I did an exhaustive path analysis. In the event that I did not conduct an exhaustive path analysis, it would still be important to analyze the branches of the program. In this program, we have 3 points where there are 2 possible branches. There are thus 6 different types of inputs that must exist to trigger all 6 possible directions. Of course, it would be important to test extreme values, end cases, and threshold values to fully ensure the branches are bug free.

Statement Static Analysis: Again, under the assumption that an exhaustive path analysis has already been conducted, statement analysis is redundant as all statements would have already been explored. I would, in the event that I could not conduct an exhaustive path analysis, start at the top of my Control Flow Diagram and see if each statement produces the desired effect. For example, the first statement of the given program is to generate an array. Well, is that array initialized? What is the access modifier? Is it final? These are the types of questions I would ask myself while moving statement by statement. In the event that something doesn't make sense, a bug may exist.

Question 5:
a) There is one glaringly obvious bug in this code. Supposedly, when an integer char is entered, the output string is supposed to contain the same char. However in the code provided, this value is doubled. Doubling the char not only produces an incorrect output value, it also can be problematic in that the integers 5-9 become 2-digit integers (2 chars) when doubled. In the code that I have provided, I do not let errors arise due to the multiplication of the values 5-9, by insisting that just a single char can be added to the output string at all times. However, if I had not designed my program like I did, it would crash the program to try and put a single char into an array when in fact the value being 'put in' is 2 chars.

Another bug is that there is no mechanism for dealing with chars that are not numeric or alphabetic. The special chars ($,#,%, etc) would disrupt this program radically. As the test cases do not contain any of the special chars, I wrote my solution to be as simple as possible, and omitted the possibility of the special chars being inputted. If a special char were to be inputted to my program, or the one provided, the program would crash.

b) Explanation of my Delta Debugging Program
In the source code that follows is a class called Question5 that is instantiated in main and has its primary display method called displayMyWork() called. Once this method is called, the input strings we are required to test are put into a list of char arrays. The inputs are printed to SYSOUT. The inputs are then run through the original processing algorithm, which has the bug of doubling integer values. I then run the inputs through a corrected version of the processing algorithm, and print to SYSOUT what the input should be. At this point, delta debugging begins.

The approach I used for my delta debugging algorithm was to compare the output from the original processing algorithm with the output from corrected processing algorithm. The algorithm works by checking if the first char of the original output matches the first char of the correct output. If the chars are not equal, we have found the first char that generates problematic output. If the first two chars are the same, the delta debugging algorithm reduce_input_so_that_bug_is_minimal() gets called with the input, excluding the first chars. The first problematic char is printed to the SYSOUT.

SOURCE CODE:

Instructions: Open a new java project.
Copy my main source code into the main program.
Create a new class called Question5 and copy this source code.

Main

```java
public class Main {
    public static void main(String[] args) {
        Question5 q5 = new Question5();
        q5.displayMyWork();
    }
}
```

```java
import java.util.*;
public class Question5 {
        public static List<char[]> inputStrings;
        static List<char[]> processedStrings;
        static List<char[]> processedStrings_correct;
        public void displayMyWork() {
                getInput("abcdefG1","CCDDEExy","1234567b", "8665");                     // insert the strings
given into a list of inputs
                welcomeMessage();                                                        // welcome the user
                processStrings();                                                        // run the input through the
processing algorithm
                printStrings(processedStrings);                                          // display the processed
strings
                messageToUser();                                                         // update the user about the
incorrectness of the processing algorithm
                processStringsCorrectly();                                               // run the input through a
corrected version of the processing algorithm
                printStrings(processedStrings_correct);                                  // display the correctly
processed strings
                messageAboutDeltaDebugging();                                            // prepare the user for
delta debugging
                deltaDebugging(processedStrings,processedStrings_correct);                      // find a
minimal bug, if it exists, for each input string
                exitMessage();                                                           // gracefully exit the program
        }
        /** IMPORTANT NOTE
         * The numeric values 5-9 when multiplied by 2 yield integers that are 2 chars long, not one.
This is the key bug.
         * Thus, the corrected version of the algorithm does not multiply integer values by 2. The
integers are simply left alone, and added to the string.
```

```
     * To conduct delta debugging, i generate a 'possiblyIncorrect' out using the given algorithm,
and a 'correct' output using the corrected version
     * of the original algorithm.
     */
    private static String printInput(char[] someChars){
        String current = "";
        for (char c : someChars){
            current = current + c;
        }
        return current;
    }
    private static void deltaDebugging(List<char[]> possiblyIncorrectOutput, List<char[]>
correctOutput){
        int index = 0;
        for (char[] inputString : possiblyIncorrectOutput) {
            System.out.println();
            System.out.println("Delta debugging for the input: " +
printInput(inputStrings.get(index)));
            System.out.println("Output was: " + printInput(inputString));
            // check if possibly incorrect output is actually correct
            if ( stringsMatch(possiblyIncorrectOutput.get(index),correctOutput.get(index))){
                System.out.println(printInput(inputString) +" is the correct output. No need to
perform delta debugging.");
            }
            else {
                reduce_input_so_that_bug_is_minimal(inputString,correctOutput.get(index),
index, 0);
            }
            index++;
        }
    }
    private static void reduce_input_so_that_bug_is_minimal(char[] possiblyIncorrectOutput, char[]
correctOutput, int index, int positionOfChar) {
        if (possiblyIncorrectOutput.length == 0 ){
        }
        else if (possiblyIncorrectOutput[0] != correctOutput[0]) {
            System.out.println("We have discovered a minimal bug for this input. It is the char: "
+ getProblematicChar(index, positionOfChar)); // get the INPUT AT INDEX);
        }
        else {
            // build the same array but take away first element
            char[] shrunkenOutput_possiblyIncorrect = new char[possiblyIncorrectOutput.length-
1];
            char[] shrunkenOutput_Correct = new char[possiblyIncorrectOutput.length-1];
            for (int i = 1 ; i < possiblyIncorrectOutput.length; i++ ) {
                shrunkenOutput_possiblyIncorrect[i-1] = possiblyIncorrectOutput[i];
                shrunkenOutput_Correct[i-1] = correctOutput[i];
            }
            reduce_input_so_that_bug_is_minimal(shrunkenOutput_possiblyIncorrect,
shrunkenOutput_Correct, index, positionOfChar+1);
        }
    }
    private static char getProblematicChar(int ithWord, int ithChar){
        // get the character from ith word in the array that is breaking the code!
        return inputStrings.get(ithWord)[ithChar];
    }
```

```
	private static void welcomeMessage(){
		System.out.println("This is Question 5 of Assignment 1 for COSC 3P95");
		System.out.println("The test cases provided in the assignment are: ");
		printStrings(inputStrings);
		System.out.println();
	}
	/** getInput(String ... s1, s2, s3,...,sn
	 * getInput accepts an arbitrary number of strings which will
	 * be tested by the processString algorithm for bugs.
	 *
	 * getInput works by converting each string into a char array
	 * the char arrays are then stored in a linked list, which is returned
	 * via a pointer to the head
	 * @param strings
	 */
	public static void getInput(String ... strings){
		List<char[]> inputList = new ArrayList<char[]>();
		for (String s : strings){
			// make the string into a char list
			char[] theStringAsACharList = s.toCharArray();
			inputList.add(theStringAsACharList);
		}
		inputStrings = inputList;
	}
	/** processStrings(char[] theString)
	 * for all strings provided in the input,
	 * process the string
	 * place processed string into a list of strings
	 */
	private static void processStrings(){
		System.out.println("I will now run the input strings through the original (buggy) algorithm, 
and print the results");
		processedStrings = processInputString(inputStrings);
		System.out.println("The input has been processed. Displaying the results now.");
	}
	private static List<char[]> processInputString(List<char[]> in){
		List<char[]> out = new ArrayList<>();
		// for each char[] in the list (for each string that was given in the input
		for (char[] inputString: in){
			char[] processedInput = new char[inputString.length];	// build a char array to hold 
the new string of chars
			int index = 0;
			for(char aChar: inputString){	// for each of the characters in the string
				if(Character.isUpperCase(aChar)){
					processedInput[index]=Character.toLowerCase(aChar);
				}
				else if(character_Is_a_Numerical_Digit(aChar)) {
					// convert aChar into its integer value
					int temp = Integer.parseInt(Character.toString(aChar));
					// double temp
					temp = temp*2;
					// convert temp back into a char
					String curr = Integer.toString(temp);
					char char_of_curr = curr.charAt(0);
					// insert the value into processedInput
					processedInput[index] = char_of_curr;
```

```
                }
                else {
                    processedInput[index] = Character.toUpperCase(aChar);
                }
                index++;
            }
            // add the processedInput to the list
            out.add(processedInput);
        }
        return out;
    }
    public static List<char[]> processStringCorrectly(List<char[]> in){
        List<char[]> out = new ArrayList<>();
        // for each char[] in the list (for each string that was given in the input
        for (char[] inputString: in){
            char[] processedInput = new char[inputString.length];       // build a char array to hold 
the new string of chars
            int index = 0;
            for(char aChar: inputString){       // for each of the characters in the string
                if(Character.isUpperCase(aChar)){
                    processedInput[index]=Character.toLowerCase(aChar);
                }
                else if(character_Is_a_Numerical_Digit(aChar)) {
                    // convert aChar into its integer value
                    int temp = Integer.parseInt(Character.toString(aChar));
                    // convert temp back into a char
                    String curr = Integer.toString(temp);
                    char char_of_curr = curr.charAt(0);
                    // insert the value into processedInput
                    processedInput[index] = char_of_curr;
                }
                else {
                    processedInput[index] = Character.toUpperCase(aChar);
                }
                index++;
            }
            // add the processedInput to the list
            out.add(processedInput);
        }
        return out;
    }
    private static void printStrings(List<char[]> in){
        System.out.println();
        for (char[] c: in){
            for (char cc : c){
                System.out.print(cc);
            }
            System.out.print("   ");
        }
        System.out.println();
    }
    private static void messageAboutDeltaDebugging(){
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println("We will now perform delta debugging.");
        System.out.println();
        System.out.println();
        System.out.println();
    }
    private static void messageToUser(){
        System.out.println("The processed strings are not as they should be. We are told in the 
assignment that numerical values should remain unchanged. \n" +
                "We clearly see that numerical values are being modified.");
        System.out.println("To perform delta debugging, we must know what the desired output 
should be.");
        System.out.println("To generate the desired outputs, I have modified the processing 
algorithm so that it only produces correct ouput, \n which we will need to conduct comparisons");
        System.out.println("The correct output is: ");
    }
    private static void exitMessage(){
        System.out.println();
        System.out.println();
        System.out.println("I hope you have enjoyed watching this program demonstrate delta 
debugging.");
        System.out.println("The program is now complete.");
        System.out.println("Thanks, bye.");
    }
    private static void processStringsCorrectly(){
        processedStrings_correct = processStringCorrectly(inputStrings);
    }
    private static boolean character_Is_a_Numerical_Digit(char c){
        if (c == '0' | c =='1'|c=='2'|c=='3'|c=='4'|c=='5'|c=='6'|c=='7'|c=='8'|c=='9'){
            return true;
        }
        else {
            return false;
        }
    }
    private static boolean stringsMatch(char[] possiblyIncorrectOutput, char[] correctOutput){
        boolean flag = true;
        for ( int j = 0 ; j < possiblyIncorrectOutput.length ; j++) {
            if(possiblyIncorrectOutput[j] != correctOutput[j]){
                flag = false;
            }
        }
        return flag;
    }
}
```