David Martin

6995948

dm20zo@brocku.ca

COSC 3P95- Assignment 1

1.

Sound: An analysis is considered sound if it has the ability to not falsely report any bugs or vulnerabilities within a program. If a sound analysis detected bugs or vulnerabilities, those anomalies are actually present within the program. In other words, a Sound Analysis is a tool that does not report any false positives.

Complete: An analysis is considered complete if it the analysis can accurately detect bugs or vulnerabilities within a program. A Complete analysis will always be report all bugs within a software. In other words, a Complete analysis is a tool that can detect all bugs and vulnerabilities that exist in the software.

➔ The main difference between a Sound and a Complete analysis is that a Complete analysis will have the ability to determine if there are bugs with certainty while a Sound analysis strictly ensures that no bugs can be falsely diagnosed in a program.

True Positive: A True Positive is when an analysis correctly identifies bugs or vulnerabilities within the software. In other words, it means the analysis has made accurate results.

True Negative: A true negative is when an analysis correctly identifies that no bugs or vulnerabilities exist within the software. Thus, it means that it determines that there is no potential bugs or vulnerabilities in the software.

False Positive: A False Positive is when an analysis has a tool that reports bugs or vulnerabilities within a program when non exists. In this case, the analysis has falsely predicted bugs within the software.

False Negative: Contrary to a False Positive, a False Negative is when a program analysis would fail to repot any vulnerabilities or bugs within a software when there are anomlilies that exist within the software.

David Martin

6995948

dm20zo@brocku.ca

➔ Changing the definition of "positive" means when finding a bug would mean that a True positive is an analysis that does have a bug. A True Negative analysis would mean that the software is free of bugs. A False Positive would mean that a bug has been falsely diagnosed in the code. A False negative would mean that an analysis would fail to report a bug in the code when a bug truly exists in the software.

➔ Changing the definition of "positive" to meaning "not finding a bug" would change the following definitions:

- True Positive: A True Positive in this case would mean that an analysis would correctly identify the fact that there are no bugs or vulnerabilities within the software.
- True Negative: A True Negative in this case would mean that the analysis has successfully encountered bugs. In other words, it means that the analysis tool has correctly detected bugs in the software.
- False Positive A False Positive in this case would mean that a program analysis would fail to report any bugs or vulnerabilities within the software when there are bugs within the software.
- True Positive: A True Positive in this case would mean that a analysis tool correctly identifies that there is no bugs or vulnerabilities within the software. In other words, it means that there is no bugs in the software.

2.

A)

a) The source code file I wrote in python contains bubble sort and the random testing case generator.

b) The sorting algorithm that was used was bubble sort as presented in the source code file. The random test case generator strictly generates random arrays with a length between 0-10 and values in each index from 0-1000. The test case generator loops through 10 different test cases meaning that 10 random arrays of random lengths each with random elements in them. After each array is sorted, the program than proceeds to see if each arrays is sorted and to see if each test case is passed.

c) Comments are in the source Code

David Martin

6995948

dm20zo@brocku.ca

d) Instructions to run the code:

→ Using a bash terminal, cd into the directory the python file.

The Python file is called randomTest.py.

→Then, run the following command:

Python3 randomTest.py

Then the following output should appear:

```
$ python3 randomTest.py
Bubble Sort Testing
Input Array :
783 958
Output Array:
783 958
The Test Case Was Passed
Input Array :
741 701 648
Output Array:
648 701 741
The Test Case Was Passed
Input Array :
41 900
Output Array:
41 900
The Test Case Was Passed
Input Array :
501 918 958 459 7
Output Array:
7 459 501 918 958
The Test Case Was Passed
Input Array :

Output Array:

The Test Case Was Passed
Input Array :

Output Array:

The Test Case Was Passed
Input Array :
535 458 743 177 542 489 840
Output Array:
177 458 489 535 542 743 840
The Test Case Was Passed
Input Array :
623 159 423 98 190 30 847
Output Array:
30 98 159 190 423 623 847
The Test Case Was Passed
Input Array :

Output Array:

The Test Case Was Passed
Input Array :
525 645 360 42 765 757
Output Array:
42 360 525 645 757 765
The Test Case Was Passed
```

e) logs are generated with print statements printing both the input array, and output array each time the program is running.

David Martin

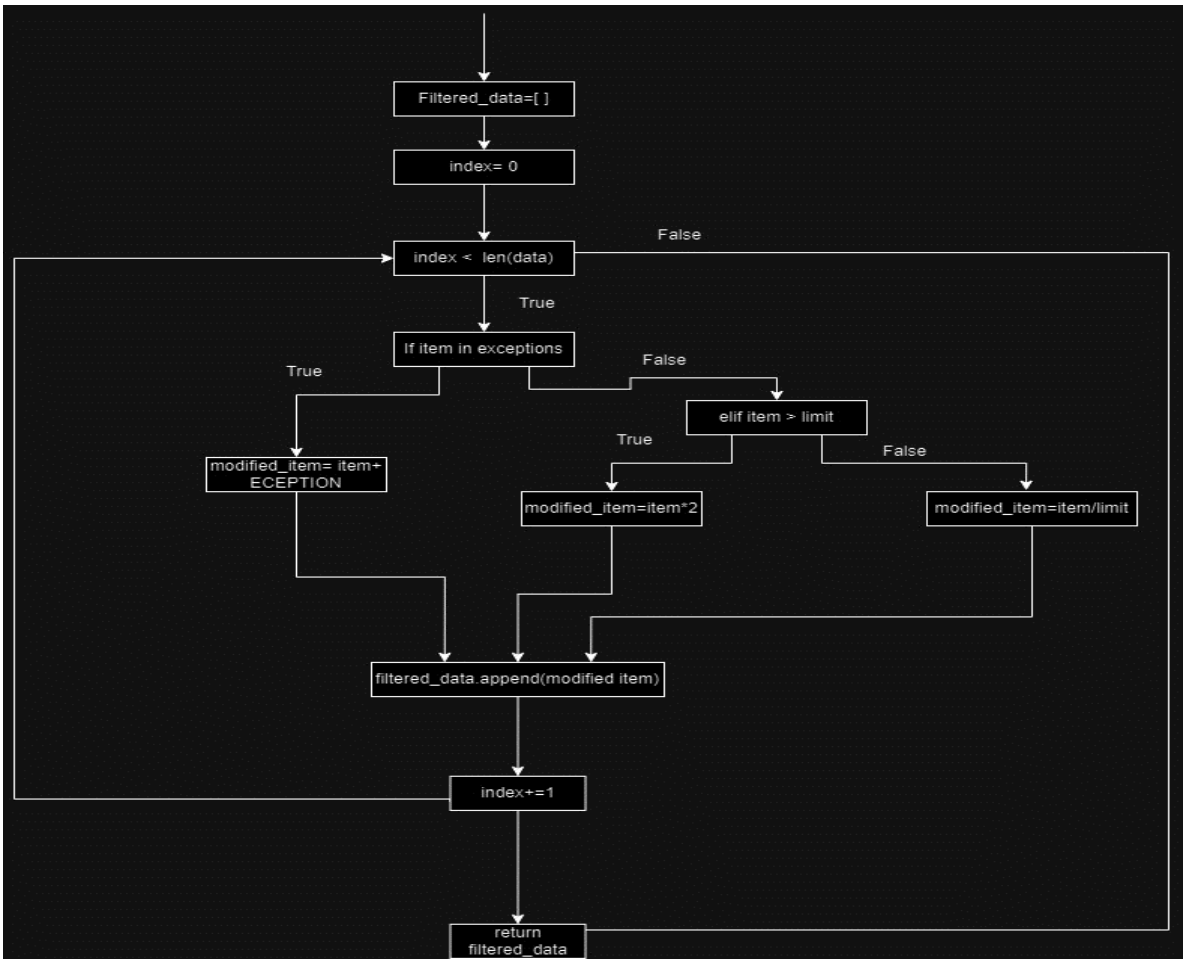
6995948

dm20zo@brocku.ca

2 B)??

3 A)

CONTROL FLOW GRAPH:



3 B)

➔ Looking at the Code provided, the filterData function takes in the following parameters: an array called data, and numerical value called “limit.” And the parameter exceptions that handles error handling. With the following parameters, we can generate random

David Martin

6995948

dm20zo@brocku.ca

inputs for data limit and exceptions. We can generate an array if random size for the data and then randomly generate the data array with either integers or float values. Then, we can randomly generate the value of the limit which can be either an integer or a float value. With these randomly generate input values, we can execute the filterData function for each test case multiple times and compare the outputs. From implementing random testing, we can compare outputs and test out each of the cases where there are elements in exceptions, cases in which item > limit and where item < limit to ensure full code coverage via random testing. Additionally, with random testing we can diagnose potential issues with the code by recording each test case.

4. A)

```
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
```

In terms of Code coverage, there are 11 statements, and three different

Test Case 1:

Data = [1,2,2,] # An array of twos and ones.

David Martin

6995948

dm20zo@brocku.ca

limit = 2

exceptions = [“Exceptions”]

{filterData(data, 2, exceptions) }

The first test case would only hit the else case in which each element in data less than or equal to the limit thus only hitting one branch which is the single else statement meaning that modified_item = item /limit would be assigned. The branch coverage would be 33% or (1/3) in this case as the if and elif branches were not met. The statement coverage would be 10/12 or %83 Statement Coverage.

Branch Coverage: 1/3 (%33)

Statement Coverage: 10/12 (%83)

Test Case 2:

Data = [“Exception”, 2, 3 ,4]

limit = 4

exceptions = [“Exception”]

{filterData(data, 4, exceptions) }

The Second test case would cover two branches in with the following input of the data array. The first if statement: “if item in exceptions” would be satisfied and modified_item= item + “_EXCEPTION” would be covered. Additionally, the integers: 2,3,4 being all less than or equal to the limit, the third else statement would be covered by that input meaning that for some indexes in data, modified_item= item/limit would be covered. Thus, in this case the branch coverage would be 2/3 or 66% and the statement coverage would be 11/12 or 91% statement coverage.

Branch Coverage: 2/3 (%66)

Statement Coverage: 11/12 (%91)

Test Case 3:

Data = [“Exception”, 2, 3 ,4]

David Martin

6995948

dm20zo@brocku.ca

limit = 3

exceptions =[“Exception”]

{filterData(data, 3, exceptions)}

The Third test case would cover all three branches within the code. In data’s first index which is an exception, will cover the first branch of the code. Thus, if item in exceptions would a branch that would be covered meaning that modified_item= item + “_EXCEPTION” would be covered. For the second and third indexes of the Data array, the else branch would be covered meaning that item= item/limit line would also be covered for this case. Lastly, in the last index of the data array, the index being 4 which is greater than the limit, would cover the branch “if item > limit ” meaning that the statement “modified_item = item * 2 ” would also be covered. Thus, in this test case 3/3 (%100) branches are covered and 12/12 (%100) statements are covered.

Branch Coverage: 3/3 (%100)

Statement Coverage: 12/12 (%100)

Test Case 4:

Data: [2,2,2]

Limit: = 2

Exceptions = [“Exception”]

{filterData(data, 2, exceptions) }

The fourth test case would only cover the else branch within the while loop. Due to the fact that all the elements in the data array are 2’s, it is equivalent to the limit. Thus, because each index in data is less than the limit. Only the else branch will be covered meaning that modified _item = item/limit would be the only statement covered within the while loop of the code. Thus, the branch coverage is 1/3 (% 33) and the statement coverage would be 10/12 (% 83)

Branch Coverage: 1/3 % 33 Statement Coverage: 10 /12

4 B)

Mutation 1:

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

David Martin

6995948

dm20zo@brocku.ca

```
    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
```

- ➔ Original Line: elif item > limit
- ➔ Mutated Lined: elif item >= limit

Mutation 2:

```
➔ def filterData(data, limit, exceptions):
➔
            filtered_data = []
➔
            index = 0

            while index > 0:
➔
                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
```

- ➔ Original Line: while index < len(data):
- ➔ Mutated Line: while index >= 0:

Mutation 3:

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

    filtered_data = []

    index = 0

    while index < len(data):

        item = data[index]

        if item in exceptions:
            modified_item = item + "_EXCEPTION"
```

David Martin

6995948

dm20zo@brocku.ca

```
        elif item > limit:

            modified_item = item * 2

    else:
            modified_item = item * limit

    filtered_data.append(modified_item)
    index += 1

    return filtered_data
```

➔ Original Line: modified_item = item/limit

➔ Mutated Line: modified_item = item * limit

Mutation 4:

```
            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 ++

    return filtered_data
```

➔ Original Line: elif item > limit.

➔ Mutated Line: if item> limit.

Mutation 5:

```
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
```

➔ Original Line: if item in in exceptions:

David Martin

6995948

dm20zo@brocku.ca

➔ Mutated Line: if item not in exceptions:

Mutation 6:

```
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
```

➔ Original Line: modified_item = item * 2
➔ Mutated Line: modified_item= item + 2

4 C)

Mutated Code:

The Code Below has 6 mutations that make it differ from the original code.

```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0

    while index >= 0:

        item = data[index]

        if item not in exceptions:
            modified_item = item + "_EXCEPTION"

        if item >= limit:

            modified_item = item + 2

        else:
            modified_item = item * limit

    filtered_data.append(modified_item)

    index +=1

    return filtered_data
```

Test Case 1: Data = [1,2,2,] # An array of twos and ones.

limit = 2

exceptions = [“Exceptions”]

David Martin

6995948

dm20zo@brocku.ca

{filterData (data, 2, exceptions)}

- ➔ The mutation first detected is the while loop as it is infinite. The condition for the while loop index >=0 will run forever, and the test case detects that mutant. Thus, test case 1 has killed 1/6 mutants.
- ➔ The second Boolean mutation is "if item not in exceptions" then we would still append "EXCEPTIONS" to the item despite not being contained in the exceptions list which is not the desired outcome of the code. Because data does not contain "exceptions" but still appends exceptions to the elements in the filtered data array means that this mutation has also been killed. Thus, test case 1 kills 2/6 mutations.
- ➔ The third mutation being the elif index > limit to the mutated version if index > limit. This mutant is detected due to the fact that the branch is still convered due to the fact that it is no longer nested and test case 1 would cover that branch when in the original version of the code it shouldn't. Thus, test case kills 3/6 mutations.
- ➔ The fourth Boolean mutation is also detected in the " if item>=limit" that was originally " if item>limit ." This is because there is the value 2 in the data array and limit = 2. Initially, this branch would not be covered but now that the mutation >= was there, that branch would be covered which shouldn't happen. Thus test case 1 has 4/6 mutations detected.
- ➔ The fifth arithmetic mutation would also be detected in this case. Because the previous mutation allows for test case 1 to enter the mutated statement " modified_item = item + 2" when it was initially " modified_item = item * 2" Thus being incorrect results for all elements greater than or equal to the limit. Thus, test case 1 has 5/6 mutations detected.
- ➔ The sixth arithmetic mutation would also be detected in this case. For elements less than the limit that would satisfy the else condition that being 1, the arithmetic mutation " modified_item= item* limit" which was initially "modified_item= item/ limit" thus bringing incorrect results for elements less than the limit in data. Thus, testcase 1 6/6 mutations detected.
    - Mutation score = 6/6 100%
    - Thus, Test Case 1 is a highly effective test case.

David Martin

6995948

dm20zo@brocku.ca

Test Case 2:

Data = [“Exception”, 2, 3 ,4]

limit = 4

exceptions = [“Exception”]

{filterData (data, 4, exceptions)}

- ➔ The mutation first detected is the while loop as it is infinite. The condition for the while loop index >=0 will run forever, and the test case detects that mutant. Thus, test case 2 has killed 1/6 mutants.
- ➔ The second mutation that would be caught is the Boolean mutant “if item not in exceptions” which was initially “if item in exceptions.” One way in which this mutant would be caught is the first index in the data array being “Exception” which is an element that is in the exceptions list would not have “EXCEPTION” appended to it when it should. Thus, Test case 2 has 2/6 mutants killed.
- ➔ The third mutation being the “elif index > limit” to the mutated version of “index > limit”. This mutant is detected due to the fact that the branch is still covered due to the fact that it is no longer nested and test case 2 would cover that branch when in the original version of the code it shouldn’t. Thus, test case kills 3/6 mutations.
- ➔ The fourth Boolean mutation is also detected in the “ if item>=limit” that was originally “ if item > limit .” This is because there is the value 4 in the data array and limit = 4 Initially, this branch would not be covered but now that the mutation >= was there, that branch would be covered which shouldn’t happen. Thus, test case 2 has 4/6 mutations detected.
- ➔ The fifth arithmetic mutation would also be detected in this case. Because the previous mutation allows for test case 2 to enter the mutated statement “modified_item = item + 2” when it was initially “ modified_item = item * 2” Thus being incorrect results for all elements greater than or equal to the limit. Thus, test case 2 has 5/6 mutations detected.
- ➔ The sixth arithmetic mutation would also be detected in this case. For elements less than the limit that would satisfy the else condition that being 2 and 3, the arithmetic mutation “modified_item= item* limit” which was initially “modified_item= item/ limit” thus bringing incorrect results for elements less than the limit in data. Thus, testcase 2 6/6 mutations detected.
    - Mutation Score = 6/6 100%
    - Thus, Testcase 2 is highly effective.

David Martin

6995948

dm20zo@brocku.ca

Test Case 3:

Data = [“Exception”, 2, 3 ,4]

limit = 3

exceptions =[“Exception”]

{filterData(data, 3, exceptions)}

- ➔ The mutation first detected is the while loop as it is infinite. The condition for the while loop index >=0 will run forever, and the test case detects that mutant. Thus, test case 3 has killed 1/6 mutants.
- ➔ The second mutation that would be caught is the Boolean mutant “if item not in exceptions” which was initially “if item in exceptions.” One way in which this mutant would be caught is the first index in the data array being “Exception” which is an element that is in the exceptions list would not have “EXCEPTION” appended to it when it should. Thus, Test case 2 has 2/6 mutants killed.
- ➔ The third mutation being the “elif index > limit” to the mutated version of “index > limit”. This mutant is detected due to the fact that the branch is still covered due to the fact that it is no longer nested and test case 3 would cover that branch when in the original version of the code it shouldn’t for all indexes of the data array. Thus, test case kills 3/6 mutations.
- ➔ The fourth Boolean mutation is also detected in the “ if item>=limit” that was originally “ if item > limit .” This is because there is the value 3 in the data array and limit = 3 Initially, this branch would not be covered on this index, but now that the mutation >= was there, that branch would be covered which shouldn’t happen. Thus, test case 3 has 4/6 mutations detected.
- ➔ The fifth arithmetic mutation would also be detected in this case. Because the previous mutation allows for test case 2 to enter the mutated statement “modified_item = item + 2” when it was initially “ modified_item = item * 2” Thus being incorrect results for all elements greater than or equal to the limit. Thus, test case 3 has 5/6 mutations detected.
- ➔ The sixth arithmetic mutation would also be detected in this case. For elements less than the limit that would satisfy the else condition that being 2 , the arithmetic mutation “modified_item= item* limit” which was initially “modified_item= item/ limit” thus bringing incorrect results for elements less than the limit in data. Thus, testcase 3 6/6 mutations detected.
    - Mutation Score 6/6 % 100
    - Thus, Testcase 3 is highly effective.

David Martin

6995948

dm20zo@brocku.ca

Test Case 4:

Data: [2,2,2]

Limit: = 2

Exceptions = [“Exception”]

{filterData(data, 2, exceptions) }

- ➔ The mutation first detected is the while loop as it is infinite. The condition for the while loop index >=0 will run forever, and the test case detects that mutant. Thus, test case 3 has killed 1/6 mutants.
- ➔ The second Boolean mutation is “if item not in exceptions” then we would still append “EXCEPTIONS” to the item despite not being contained in the exceptions list which is not the desired outcome of the code. Because data does not contain “exceptions” but still appends exceptions to the elements in the filtered data array means that this mutation has also been killed. Thus, test case 4 kills 2/6 mutations.
- ➔ The third mutation being the “elif index > limit” to the mutated version of “index > limit”. This mutant is detected due to the fact that the branch is still covered due to the fact that it is no longer nested and test case 4 would cover that branch when in the original version of the code it shouldn’t for all indexes of the data array. Thus, test case 4 kills 3/6 mutations.
- ➔ The fourth mutation in this case would also be detected. This is due to the fact that the data array contains 2 which is also equal to the limit being 2. Initially this branch would not have been covered, but due to the Boolean mutation were the condition was initially “ if item > limit ” become “if item >= limit ,” this branch would be covered when it shouldn’t in this case. Thus, Testcase 4 kills 4/6 mutations.
- ➔ The fifth arithmetic mutation would also be detected in this case. Because the previous mutation allows for test case 2 to enter the mutated statement “modified_item = item + 2” when it was initially “ modified_item = item * 2” Thus being incorrect results for all elements greater than or equal to the limit. Thus, test case 3 has 5/6 mutations detected.
- ➔ In this case, the arithmetic mutation modified_item= item* limit” which was initially “modified_item= item/ limit” would not get caught for this test case. This is due to the fact that each element in the array is equal to the limit. Thus, there is no case in which the else branch to the condition if item >= limit would be hit. Thus, Test case 4 does not detect this mutation.
    - Mutation score = 5/6 %83
    - Thus, test case 4 is effective but misses a mutation

David Martin

6995948

dm20zo@brocku.ca

Test Cases ranked (Worst to Best):

1: Testcase 4, 2 : Testcase 2, 3:Testcase 1, 4: Testcase 3 (As it has 100% code coverage)

4 D)

BRANCH TESTING:

- When conducting static analysis of the code above in the form of branch testing, it would be wise to provide A test case in which all branches would be covered. In other words, each branch is tested at least once. A testcase I would provide to ensure that each branch is testing and there is %100 statement coverage is the following example:
  - Data= [4,3,2,1], exceptions = [2], limit = 3
  - Thus each branch would be covered by the following input including:
    - If item in exceptions would be covered.
    - Elif item > limit would be covered
    - Else: would be covered
  - Thus each condition would be covered ensure % 100 branch coverage .

PATH TESTING:

- When conducting a static analysis of the code above in the form of path Testing. It would be a wise to make a control flow graph to observe all of the possible paths in the code. That way the logic of each branch could be observed and the tester can identify each independent path that can be taken within the code. After finding each linear path that exists in the code, it would be good to write test cases that would take each path in the code. An example of a test case that does this is the following test case:
  - Data= [4,3,2,1], exceptions = [2], limit = 3
  - Thus each branch would be covered by the following input including:
    - If item in exceptions path would be visited
    - Elif item > limit would be visited
    - Else: would path would be visited
  - Thus each condition would be overed ensure that each individual path is executed.

STATEMENT TESTING:

- When conducting a static analysis of the code above in the form of statement testing, it would be good practice to ensure that each statement in the code would be executed at least once. To do so, it would be vital to provide a test case in which would have % 100 branch coverage, then with the information from the control flow graph, ensure that each individual path is executed. Thud, in turn each statement in the code would be

David Martin

6995948

dm20zo@brocku.ca

executed ensuring that each statement would be tested despite being in different branches or paths.

5 A)

- ➔ I implemented a random test case generator with strings containing uppers case characters and lower-case characters as well as numbers. The random test case is in the file: questionFive.py and is fully commented.
- ➔ To run the file cd into the directory with the python file and type:
  - python questionFive.py

```
$ python questionFive.py
PART A:
INPUT: G75Mr87
OUTPUT: g7755mR8877
INPUT: et3512f67B
OUTPUT: ET33551122F6677b
INPUT: u52li
OUTPUT: U5522LI
INPUT: i730
OUTPUT: I773300
INPUT: AtHTN
OUTPUT: aThtn

INPUT STRING: abcdefG1
OUTPUT STRING: ABCDEFg11
TRIMMED:
```

- ➔ Thus I was able to detect the bugs via random testing. The problem with the code is that when a character is a numerical value, that number is duplicated in the next index. The specific line causing this problem is the following line:

```
➔ elif char.isnumeric():
      output_str += char * 2
```

5 B)

I attempted to implement delta debugging by making a function called

David Martin

6995948

dm20zo@brocku.ca