# COSC 3P95 - Software Analysis & Testing
# Assignment 1

1. .

**Sound Analysis:** Sound analysis refers to the ability of an analysis tool to not report any false positives. When the tool or technique says there is a bug, it means there is indeed a bug. Soundness guarantees that all bugs which are found are indeed vulnerabilities, but cannot guarantee to find all bugs within the program.

**Complete Analysis:** Complete analysis refers to the ability of an analysis tool or technique to prevent false negatives. If there is a bug to be found, the tool or technique will find it. A complete analysis tool will report all bugs that exist within the software. Completeness guarantees that it will find all bugs within a program, but may also report false positives.

**True Positive:** A true positive is when a software analysis tool or technique finds a bug in the software, and it is determined to actually be a bug.

**True Negative:** A true negative is when a software analysis tool or technique confirms that there is no bug in the program, and that is indeed correct.

**False Positive:** A false positive refers to when a software analysis tool or technique finds a bug in the program, but is later determined to not actually be a vulnerability.

**False Negative:** A false negative refers to when a software analysis tool or technique confirms that there are no bugs in the program, but it is incorrect, and there are actually vulnerabilities that exist.

When these terms change, the positive and negative versions of true and false will swap:
**“Positive” means finding a bug:**
**True Positive:** When the tool correctly identifies a bug within the program.
**True Negative:** When the tool correctly identifies no bugs within the program.
**False Positive:** When the tool incorrectly identifies an issue within the program which is not actually a bug.
**False Negative:** When the tool incorrectly identifies no bugs within the program, but misses certain vulnerabilities.

**“Positive” means not finding a bug:**
**True Negative:** When the tool correctly identifies a bug within the program.
**True Positive:** When the tool correctly identifies no bugs within the program.
**False Negative:** When the tool incorrectly identifies an issue within the program which is not actually a bug.
**False Positive:** When the tool incorrectly identifies no bugs within the program, but misses certain vulnerabilities.

2. A.

a) See attached

b) To create a random test case generator I first created a sorting algorithm which takes any integer array as input and sorts the array in ascending order. It does this using a bubble sort, which goes through every integer in the array and checks if the integer before it is greater than itself. If it is, it swaps positions so that the array ends sorted in ascending order. Then, I created an integer array called generateRandomTestCase, which takes a range given by the user as the maximum length of the array. It also takes the max value which any integer in the array can be, specified by the user. It then generates a random number within this range, using Math.random, for the length of the array, and fills this array with random integers between 1 and the max value.

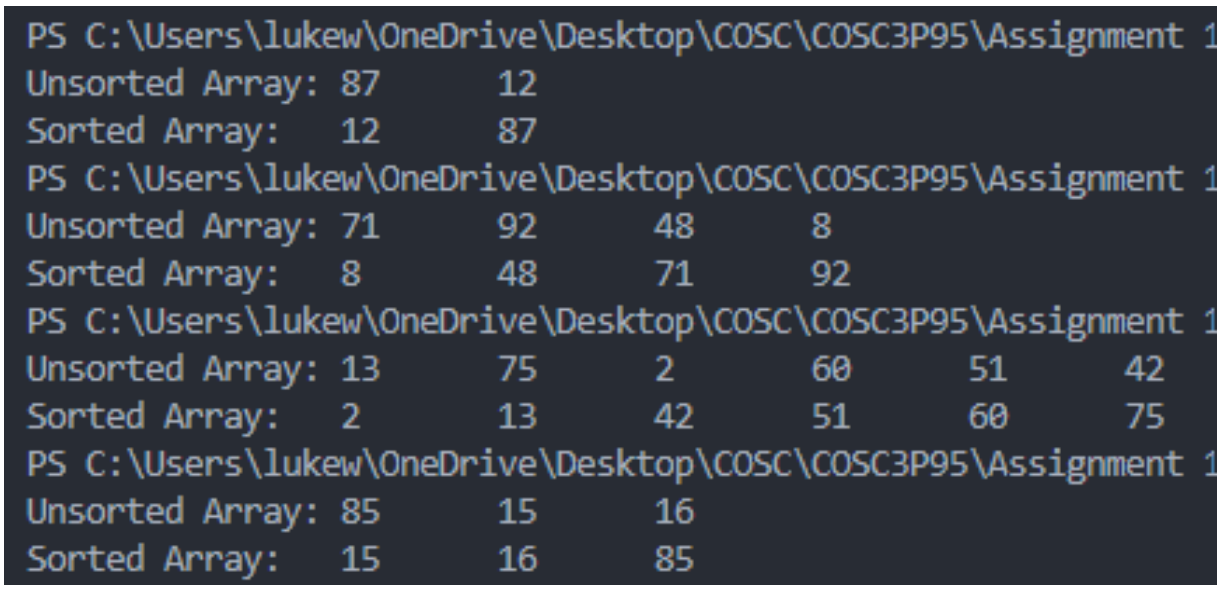
   The random test case generated found no bugs in the version submitted. However, it produced a bug in earlier versions where the last integer value of the array remained constant and was not sorted. This is because on line 42, "j<arr.length-**1**" was originally a mistake as "j=arr.length-**1**". This "1" left the last integer value unaltered.

c) See attached code.

d) To compile and run the code, set the range for the length of the random array and the max value of the random array on line 15, and press run.

e) The below screenshot shows 4 logs generated by the print statements, with the unsorted array followed by the sorted array. Each array is generated randomly, with a max length of 15, and max values within each array of 100. No bugs were found.

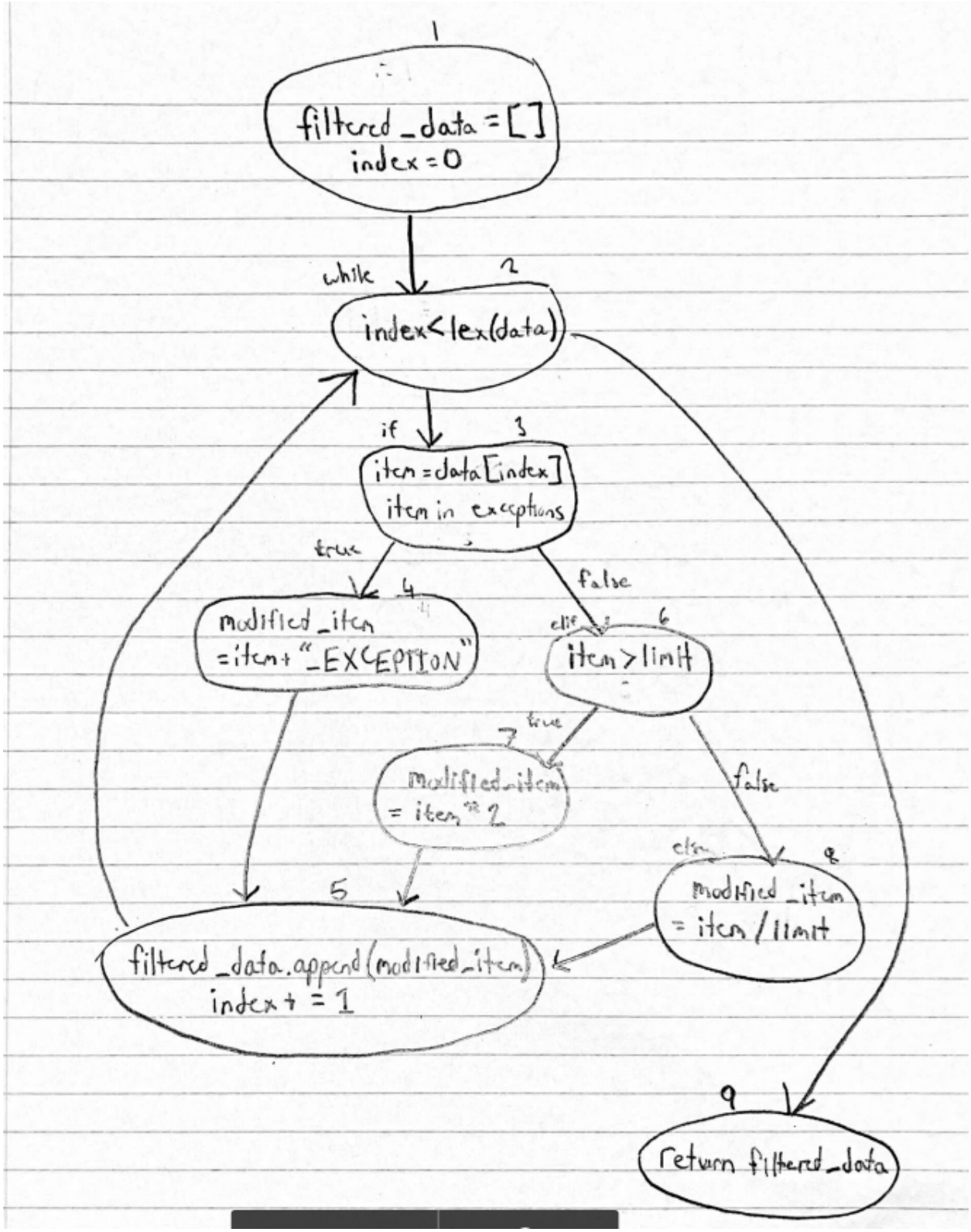
f) The below screenshot also shows logs generated by the random test executions. Each test was a pass because the unsorted array passes through the bubbleSort method and returns the correct sorted array.



B.
Using the following context free grammar, we can generate all arrays of every length containing every integer, both negative and positive.

Set A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Regular Expression: (-)(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)*  + (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)*
Production rules:
$S \rightarrow 0S \mid 1S \mid 2S \mid 3S \mid 4S \mid 5S \mid 6S \mid 7S \mid 8S \mid 9S$
$S \rightarrow \varepsilon$

3. A)


1
filtered_data = []
index = 0
while
2
index < len(data)
if
3
item = data[index]
item in exceptions
true
4
modified_item
= item + "_EXCEPTION"
false
elif
6
item > limit
true
7
modified_item
= item * 2
false
else
8
modified_item
= item / limit
5
filtered_data.append(modified_item)
index + = 1
9
return filtered_data

B) Steps for random testing the code include:

1. **Understand the code and the key variables to be tested:**

- In this case, it includes variables such as data, limit, and exceptions.
- The purpose of this code is to filter the data, based on various limits and exceptions.
- Further understand the flow of the code, which might be through a control flow diagram, such as the one above.

2. **Generate Random Test Cases:**

- Randomly generate different values for data, limit, and exceptions. This would mean to vary the data types, the length of the lists, and the numerical values of the three.
- Create cases which are uncommon, such as empty lists for data or exceptions.
- Check for boundary value errors.

3. **Execute the code with the random test cases:**

- Step through the code with the random test cases to uncover bugs and vulnerabilities.
- Understand the if-elif-else statements, and what bugs arise in their sections.

4. **Validate the results:**

- Check with the results against the expected results for each of the test cases.
- Understand various unexpected results and bugs in order to refine them.

5. **Iterate and repeat the process:**

- Modify tests and repeat the process once bugs are fixed and vulnerabilities are repaired until no bugs surface, and tests are satisfied.

4.

A) .

1. data = []
   limit = 5
   exceptions = [12]
   The code coverage for this test case is 4/14 = 28% or about 30%. It doesn't enter the while loop and covers minimal code.

2. data = [3]
   limit = 5
   Exceptions = [12]
   The code coverage for this test is 10/14 = 71%. It enters the while loop and jumps to the else statement.

3. data = [3, 7]
   limit = 5
   Exceptions = [12]
   The code coverage for this test case is 12/14 = 86%. It enters the while loop twice and jumps to both else and elif statements.

4. data = [3, 7, 12]
   limit = 5
   Exceptions = [12]
   The code coverage for this test case is 14/14 = 100%. It performs every statement within the code.

B) **Original:**

```
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
```

1. Changed item = data[index] to item = exceptions[index]

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

2. Changed if item is in exceptions to if item is not in exceptions.

```
def filterData(data, limit, exceptions):
    filtered_data = []
    index = 0
    while index < len(data):
        item = data[index]
        if item is 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
```

3. Changed elif item > limit to elif item < limit.

```
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
```

4. Changed elif item > limit to elif item == limit

```
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
```

5. Changed modified_item = item * 2 to item / 2.

```
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
```

6. Changed modified_item = item / limit to item * limit.

```
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
```

C)

| | Test 1:<br>data = []<br>limit = 5<br>exceptions=[12]<br>Filtered_data = [] | Test 2:<br>data = [3]<br>limit = 5<br>exceptions=[12]<br>filtered_data = [⅜] | Test 3:<br>data = [3, 7]<br>limit = 5<br>exceptions=[12]<br>Filtered_data = [⅜,14] | Test 4:<br>data = [3, 7, 12]<br>limit = 5<br>exceptions=[12] |
|---|---|---|---|---|
| Mutant 1:<br>Item = data[index] -> item=exceptions[index] | Pass | Fail | Fail | Fail |
| Mutant 2:<br>Item in exceptions -> Item not in exceptions | Pass | Fail | Fail | Fail |
| Mutant 3:<br>Item > limit -> Item < limit | Pass | Fail | Fail | Fail |
| Mutant 4:<br>Item > limit -> Item == limit | Pass | Pass | Fail | Fail |
| Mutant 5:<br>Item * 2 -> Item / 2 | Pass | Pass | Fail | Fail |
| Mutant 6:<br>Item / limit -> Item * limit | Pass | Fail | Fail | Fail |
| Mutation score | 0% | 67% | 100% | 100% |

I would rank these tests in the following order: test 4, test 3, test 2, test 1. This is based on their mutation score as tests 3 and 4 both have 100%, 2 has 67%, and 1 has 0%. I chose to place test 4 over test 3 because it has more code coverage, meaning all bugs should be able to be found.

D)

**Path Analysis:** Refers to the paths that can be taken throughout the code. In this case, there are four paths:

1. The data set is empty and the while loop is avoided. This results in an empty set to return.
2. If item is in exceptions.
3. ElIf item > limit.
4. Else (default)

For analysis, one would analyze the four main paths the code can take, using the tests to determine possible bugs.

**Branch Analysis:** Branch analysis refers to the different branches that are taken based on conditional statements. For this case, it is the same as the possible paths that can be taken, with if, elif, and else statements and the original while statement.

**Statement Analysis:** Statement analysis involves analyzing every statement in the code. There are 14 statements in this code to analyze for bugs and vulnerabilities.

5. .

a) The bug in this code relates to the numerical elif line. According to the question, the numeric characters are supposed to remain unchanged. However, the current version of the code repeats each numeric character a second time and adds it to the output. For example, an input string of “CH12l” would be output as “ch1122L”, which is incorrect according to the requirements. This is on line 7 or: “output_str += char * 2”. The correct code would be “output_str += char”. My strategy for finding this bug was first to understand the code and the requirements of the output, and then pass strings containing random numbers and letters to determine the bug. I noticed that only the numerical values were returning incorrectly, which led me to the elif char.isnumeric(): line.

b) Code is attached. Results for input strings below:

```
PS C:\Users\lukew\OneDrive\Desktop\COSC\COSC3P95\Assignment 1> python -
Input String:abcdefG1
The minimal value which produces a bug for the input string is:1
PS C:\Users\lukew\OneDrive\Desktop\COSC\COSC3P95\Assignment 1> python -
Input String:CCDDEExy
The minimal value which produces a bug for the input string is:y
PS C:\Users\lukew\OneDrive\Desktop\COSC\COSC3P95\Assignment 1> python -
Input String:1234567b
The minimal value which produces a bug for the input string is:1234567
PS C:\Users\lukew\OneDrive\Desktop\COSC\COSC3P95\Assignment 1> python -
Input String:8655
The minimal value which produces a bug for the input string is:8655
PS C:\Users\lukew\OneDrive\Desktop\COSC\COSC3P95\Assignment 1>
```

We know that the bug in this code is from the numeric characters being repeated when the program is called, instead of remaining the same. Because of this, this delta debugging algorithm works by checking the length of the substring before and after it is passed to the

proccessString method, and removing the substring from the result if they are equal. If they are equal, this means the substring does not produce a bug in the code.