COSC 3P95 – Software Analysis & Testing Assignment 1

For

Professor Naser Ezzati-Jivan

By

Mohit Gummaraj Kishore

mg21bi@brocku.ca

07343429

Table of Contents

Q1

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.

Topic	Definition
Sound Analysis	Sound Analysis technique refers to the ability to not provide any false positives while analysing any software. Sound analysis tools or techniques will never report a bug that does not exist in a software. In other words, using sound analysis prevents false positives. However, a sound analysis does not guarantee it will find all bugs/vulnerabilities.
Complete Analysis	Complete Analysis technique refers to the ability to detect all actual bugs that exist in a software. Complete analysis tools will detect and report all bugs. In other words, with complete analysis if there is a bug, it will be reported. Complete analysis prevents false negative. However, complete analysis does not guarantee it will not report a bug/vulnerability that does not exist.
True Positive	When a software analysis tool reports a bug that exist and the reported bug is an issue, it is known as true positive.
True Negative	When a software analysis tool reports the absence of a feature or a bug that is supposed to be in the software.
False Positive	When a software analysis tool reports a bug when there are no issues in a software, it's called false positive.
False Negative	When a software analysis tool does not report a bug when we know there is supposed to be a bug, it is known as false negative.

when "positive" means not finding a bug

Topic	Definition
True Positive	When a software analysis tool reports there are no bugs and there are actually no bugs.
True Negative	When a software analysis tool reports a bug and the bug does exist in the software.
False Positive	When a software analysis tool reports there are no bugs in the software, but the software has some bugs.
False Negative	When a software analysis tool identifies a bug, but there are no bugs in the software.

The meaning of positive can be interchanged and is important to understand the context when it comes to software analysis. Overall, the above four situations are similar, but only the terms used to identify the bugs change based on the meaning of positive.

Q2:

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) Your submission should consist of:

a. Source code files for the sorting algorithm and the random test case generator.

b. Explanation of how your method/approach works and a discussion of the results (for example, if and how the method was able to generate or find any bugs, etc.). You can also include bugs in your code and show your method is able to find the input values causing that.

c. Comments within the code for better understanding of the code.

d. Instructions for compiling and running your code.

e. Logs generated by the print statements, capturing both input array, output arrays for each run of the program.

f. Logs for the random test executions, showing if the test was a pass and the output of the execution (e.g., exception, bug message, etc.).

B) Provide a context-free grammar to generate all the possible test-cases

Q3

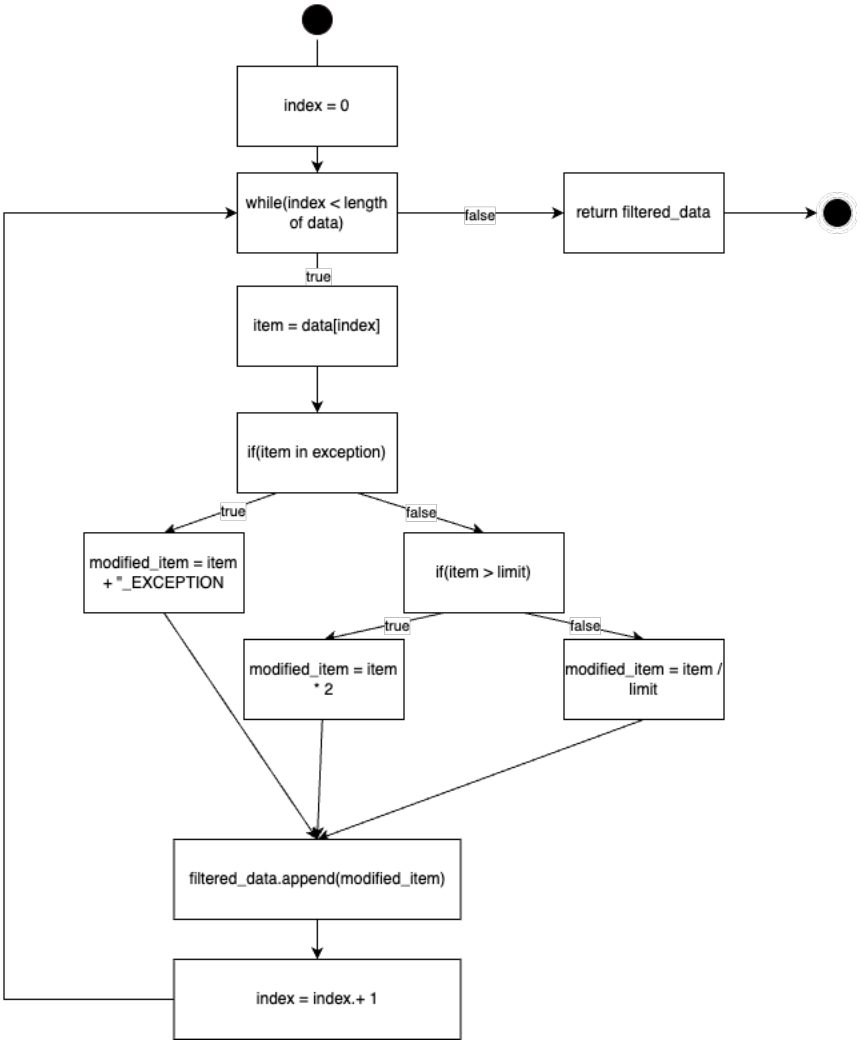
A) For the following code, manually draw a control flow graph to represent its logic and structure. The code is supposed to perform the followings:

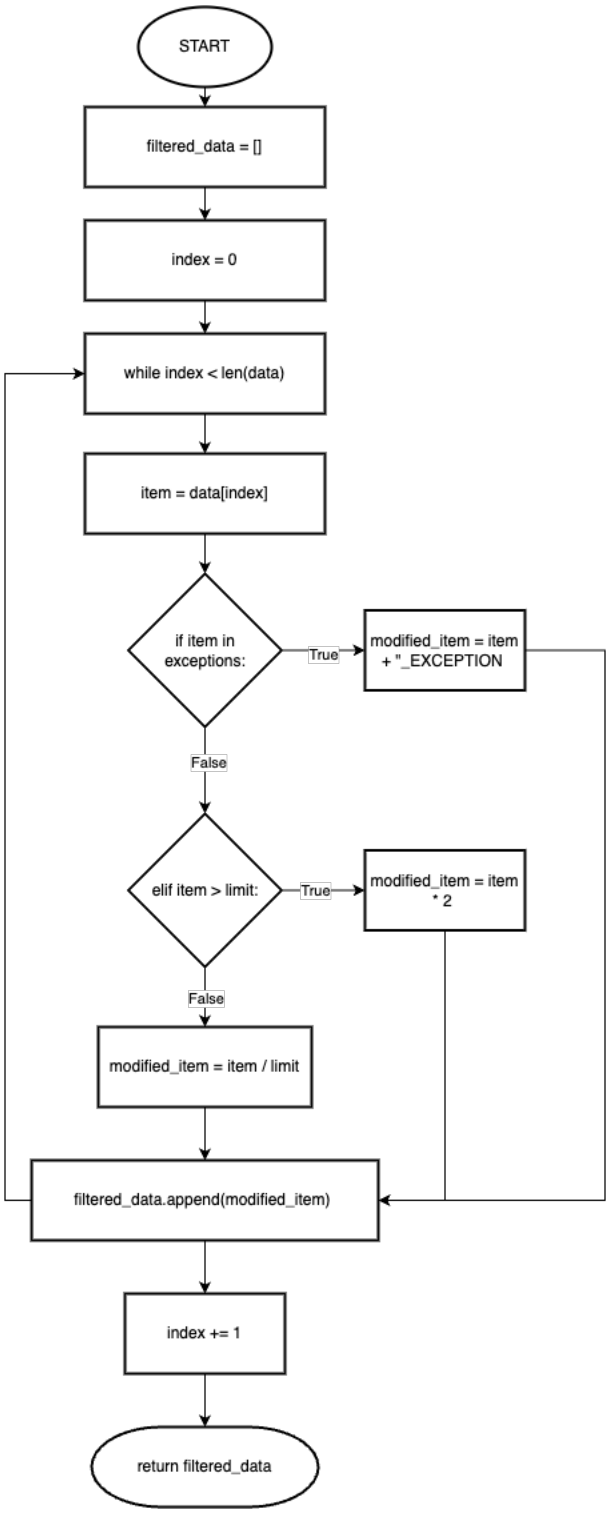
a. If an item is in the exceptions list, the function appends "_EXCEPTION" to the item.

b. If an item is greater than a given limit, the function doubles the item.

c. Otherwise, the function divides the item by 2.

Note: There was a discrepancy between the code and the above requirement in step (c). For the context free grammar diagram, I have used the code as the source.




START
filtered_data = []
index = 0
while index < len(data)
item = data[index]
if item in exceptions:
True
modified_item = item + "_EXCEPTION
False
elif item > limit:
True
modified_item = item * 2
False
modified_item = item / limit
filtered_data.append(modified_item)
index += 1
return filtered_data

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.

The goal of random testing is finding defects in a software by providing a diverse number of inputs. To test the above code, we can use the following implementation:

1. Create a function called random_filter_test
2. The method takes an input of test cases.
3. Each test case is a tuple of case number and an object called filter.
    a. Filter class attributes:
        i. Data: an array of numbers/integers
        ii. Exceptions: an array of numbers/integers
        iii. Limit: a number that specifies the limits
        iv. Expected outcome: A hardcoded array of the expected solution.
4. The function iterates n number of times, where n = length of test cases array
5. For each test case item, the loop performs the following:
    a. Execute the filter function for data, exceptions, and limit from the filter object.
    b. Checks if the output produced is the same as the expected output from the filter object.
    c. If the outputs the same, it logs a successful message for each run.
    d. If the outputs do not align, it logs an unsuccessful message.
        i. Stores the filter object and the output in an array to keep track of failed test cases
6. If the length of failed test cases is greater than zero, the test method returns a false to indicate there are bugs in the code.

Q4

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 No.	Test Case	Coverage
1	Length of the data array is 1 • data[0] is in exception	64%
2	Length of the data array is 2 • data[0] is in exception. • data[1]==limit	78%
3	Length of the data array is 3 • data[0] is in exception • data[1]==limit • data[2] > limit	100%
4	Length of the data array is 2 • data[0] is in exception • data[1] is not in exception • data[1] is < limit	64%

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

Mutations No.	Code	Mutation Explained	Type of Mutation

1	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
		Reducing the index by 1 at the end of each iteration will cause the program to stop after only 1 check.	Arithmetic Mutation

2	def filterData(data, limit, exceptions): filtered_data = []		Statement Mutation

	index = len(data)
	
	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
		Setting the index to length of the data array will stop the program from running the while loop.	

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”
	  elif item > limit:
	  modified_item = item - 2
	  else:
	  modified_item = item / limit
	
	  filtered_data.append(modified_item)
	  index += 1
	
	return filtered_data
		Changing the returned value when item is greater than limit will make the expected outcome different. This will cause the test cases to fail.	Arithmetic 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 – 1)
		Changing the returned value for the else condition will provide a different output failing test cases for the provided values.	Arithmetic Mutation

	filtered_data.append(modified_item) index += 1 return filtered_data		

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
		Checking if the item is not in exception and adding the tag exception to it at the end will provide the wrong output.	Boolean 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
		Changing the logic of item selection will create a fault in the expected output.	Boolean Mutation

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.

Mutation Score is calculated using:

- Mutation_score=killed_mutants/ total_mutants

Case No.	Mutation Code	Mutant	Explanation

Case Number: 1
Length of the data array is 1
• data[0] is in exception
 	
Score:
 	
Killed mutants = 2
Total mutants = 6
 	
Score = 2/6
Score = 1/3
	def filterData(data, limit, exceptions):
	filtered_data = []
	index = 0
	
	while index < len(data):
	item = data[index]
	if item in exceptions:
	modified_item = item + "_EXCEPTION”
	elif item > limit:
	modified_item = item * 2
	else:
	modified_item = item / limit
	
	filtered_data.append(modified_item)
	index -= 1
	
	return filtered_data
		Passed	The test case only covers for changes in the first condition.

	def filterData(data, limit, exceptions):
	filtered_data = []
	index = len(data)
	
	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
		Killed	The while loop is not executed, resulting in a failed test case

	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
		Passed	The test case only covers for changes in the first condition.

	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 – 1)
	
	filtered_data.append(modified_item)
	  index += 1
	
	return filtered_data
		Passed	The test case only covers for changes in the first condition.

	def filterData(data, limit, exceptions):
	  filtered_data = []
	  index = 0
	
	  while index < len(data):
	  item = data[index]
	  if item not in exceptions:
		Killed	The mutant is killed because the only data is in the exception list and the code does not append “_EXCEPTION”

	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		

	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
		Passed	The test case only covers for changes in the first condition.

2
 	
Length of the data array is 2
• data[0] is in exception.
• data[1]==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
		Killed	The second item is not read and the expected output is not produced.

Score Killed Mutants = 4 Total Mutants = 6 Score = 4/6 = 2/3	filtered_data.append(modified_item) **index -= 1** return filtered_data		

	def filterData(data, limit, exceptions):
	filtered_data = []
	index = len(data)
	
	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
		Killed	The while loop is not executed, and the expected output is not produced.

	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
		Passed	The expected data is produced for the above test case.

	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 – 1)
	
	filtered_data.append(modified_item)
	index += 1
	
	return filtered_data
		Killed	The expected data is not produced for data[1] == limit

	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
		killed	The mutant is killed because the only data is in the exception list and the code does not append “_EXCEPTION”

	def filterData(data, limit, exceptions):
	filtered_data = []
	index = 0
	
	while index < len(data):
	item = data[index]
	if item in exceptions:
	modified_item = item + "_EXCEPTION”
		Passed	There is no test coverage for the else if case when item < limit

	elif item < limit: modified_item = item * 2 else: modified_item = item / limit filtered_data.append(modified_item) index += 1 return filtered_data		

3
 	
Length of the data array is 3
• data[0] is in exception
• data[1]==limit
• data[2] > limit
 	
Score
 	
Killed Mutants = 6
Total Mutants = 6
 	
Score = 6/6 = 1
	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
		Killed	Only one element of the data array is executed

	def filterData(data, limit, exceptions):
	filtered_data = []
	index = len(data)
	
	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
		Killed	The while loop is not executed as index is equal to the length of data. So the output does not align with the expected output.

	filtered_data.append(modified_item) index += 1 return filtered_data		

	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
		Killed	For data[2] the output is different from the expected output

	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 – 1)
	
	filtered_data.append(modified_item
	)
	  index += 1
	
	return filtered_data
		Killed	The expected output for data[1] fails as the expected output is different.

	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
		Killed	Both data[1] and data[2] will have the prefix “_EXCEPTION” and that does not align with the expected output.

	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
		Killed	As the output for data[1] is in correct from the expected output.

4
Length of the data array is 2
	def filterData(data, limit, exceptions):
	  filtered_data = []
	  index = 0
	
	  while index < len(data):
	  item = data[index]
	  if item in exceptions:
		killed	As the array will not be process after the first item.

<table>
<tr>
<td rowspan="3">
• data[0] is in exception
• data[1] is not in exception
• data[1] is < limit

Score

Killed Mutants = 5
Total Mutants = 6

Score = 5/6
</td>
<td>
modified_item = item + "_EXCEPTION”
elif item > limit:
modified_item = item * 2
else:
modified_item = item / limit

filtered_data.append(modified_item)
<u>index -= 1</u>

return filtered_data
</td>
<td></td>
<td></td>
</tr>
<tr>
<td>
def filterData(data, limit, exceptions):
filtered_data = []
<u>index = len(data)</u>

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
</td>
<td>Killed</td>
<td>As no items from the array will be executed</td>
</tr>
<tr>
<td>
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:
<u>modified_item = item - 2</u>
else:
modified_item = item / limit
</td>
<td>Passed</td>
<td>All items will be executed as expected as modified line will not be reached</td>
</tr>
</table>

	filtered_data.append(modified_item) index += 1 return filtered_data		

	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 – 1)
	
	filtered_data.append(modified_item
	)
	  index += 1
	
	return filtered_data
		Killed	As the output for data[1] will be different from the expected output

	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
		Killed	As the output for data[0] will be different from the expected output.

	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
		Killed	The output for data[1] will be different from the expected output.

Ranking:

Ranking of the test cases is based on the mutation score. The higer the mutations score, the better the test suite and its coverage. In our case test suite 3 has the highest score as every execution kills the mutant.

Case No.	Test Case	Coverage	Score	Ranking
1	Length of the data array is 1 • data[0] is in exception	64%	1/3	4
2	Length of the data array is 2 • data[0] is in exception. • data[1]==limit	78%	2/3	3
3	Length of the data array is 3 • data[0] is in exception • data[1]==limit • data[2] > limit	100%	1	1
4	Length of the data array is 2 • data[0] is in exception • data[1] is not in exception • data[1] is < limit	64%	5/6	2

D) Discuss how you would use path, branch, and statement static analysis to evaluate/analyse the above code.

Analysis Type	Description
Path	• Path testing includes testing each and every conditions of the code • All the above test cases uses a form of path testing, • Mutations are one of the best way to ensure every path is tested • Adding Boolean mutations can help with path testing.
Branch	• The main goal of branch testing is to test every statement of the code. • Test case 3 uses a branch analysis, every condition in the while loop is executed to ensure all conditions are tested at least once.
Statement	• Statement testing includes analysing each line of code. • For this, a user will have to create test cases for 100% of the code coverage. This includes assignments, if statements and loops.

Q5

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.

Upon manually reviewing the code, and running the test on the provided test strings, I found the error was in the line "# output_str += char * 2". The code multiplied the character item with 2. This printed each item twice.

B: **Implement Delta Debugging, in your preferred programming language to minimize the input string that reveals the bug. Test your Delta Debugging code for the following input values provided.**

- **"abcdefG1"**
- **"CCDDEExy"**
- **"1234567b"**
- **"8665"**

Briefly explain your delta-debugging algorithm and its implementation and provide the

source code in/with your assignment.

I performed delta debugging using python. I simply wrote 5 lines of code to automate the testing process for identifying the bug. Once I noticed only the string with numeric were producing the wrong output, I was able to identify the specific path that caused this error. I commented out the provided line of code and added the correct line of code that aligns with the given requirements.