Yohandi 120040025 CSC4001 Assignment 2

Q1 (a) Mutant#1

(b) Mutant#2, Mutant#3, Mutant#4, Mutant#5

(c) Mutant #1, Mutant #2, Mutant #3, Mutant #4, Mutant#5

(d) Mutation score of Case#4 is 40%.

Surviving mutants are Mutant#1, Mutant#2, Mutant #4

(e) Mutation score of case #5 is 100%.

No surviving mutants

Q2 (a) Case#1, Case#3, Case #5

(b) Both Case#3 and Case#5 have 100% mutation scores, implying that each of the two cases can detect bugs of all mutants (referring to Mutant#1, Mutant #2, Mutant#3, Mutant #4, Mutant#5)

	output
$P(t)$	[1,2,3,4]
$m_1(t)$	[4,3,2,1]
$m_2(t)$	[1,1,2,2]
$m_3(t)$	-
$m_4(t)$	[4,3,2,1]
$m_5(t)$	[4]

Case#3

	output
$P(t)$	[1,2,3,4,5]
$m_1(t)$	[4,5,3,1,2]
$m_2(t)$	[2,2,2,3,3]
$m_3(t)$	-
$m_4(t)$	[5,4,3,2,1]
$m_5(t)$	[4]

Case#4

The 100% mutation scores are shown above (since $p(t) \neq m_i(t)$ for $i=1,2,\dots,5$) for both cases

Q3 (a) Assume the referred $g(x)$ follows the one in the statement, then:

x	$f(x)$	$g(x)$	$f(g(x))$ ← this is also z that you are looking for
[1]	[1]	[1]	[1]
[1,2,3,4]	[1,2,3,4]	[4,3,2,1]	[1,2,3,4]
[4,3,2,1]	[1,2,3,4]	[1,2,3,4]	[1,2,3,4]
[1,1,1,1]	[1,1,1,1]	[1,1,1,1]	[1,1,1,1]
[4,5,3,1,2]	[1,2,3,4,5]	[2,1,3,5,4]	[1,2,3,4,5]

since $y=f(x)$ and $z=f(g(x))=F(y)$

y	$F(y)$
[1]	[1]
[1,2,3,4]	[1,2,3,4]
[1,2,3,4]	[1,2,3,4]
[1,1,1,1]	[1,1,1,1]
[1,2,3,4,5]	[1,2,3,4,5]

$\Rightarrow z = F(y) = \text{Sorted}(y) = y$ (since $y=f(x)$, y is sorted)

(b)

cases	output M4(cases)	output M4(g(cases))
Case #1	[1]	[1]
Case #2	[4,3,2,1]	[4,3,2,1]
Case #3	[4,3,2,1]	[4,3,2,1]
Case #4	[1,1,1,1]	[1,1,1,1]
Case #5	[5,4,3,2,1]	[5,4,3,2,1]

No mutant detected since output for both cases and g(cases) with Mutant #4 are exactly the same

Q4. Similarities:

- Mutation-Based Fuzzing & Generation-Based Fuzzing aim to identify software vulnerabilities by providing unexpected inputs to the target application
- Both approaches use automated techniques to generate test cases
- Both methods seek to achieve high code coverage and explore as many code paths as possible

Differences:

	Mutation-Based Fuzzing	Generation-Based Fuzzing
Test Case Generation	modify existing inputs	Create from scratch using a model
Knowledge of Input Format	can operate with less	require
Test Case Diversity	limited (use seed)	wider variety
Efficiency	more efficient, but may miss vulnerabilities	less efficient, but can explore more diverse inputs
Test Result Example	crash happens when modify a seed file	crash happens when change new input format

Q5. Test case reduction helps simplify larger or complex test cases that lead to software problems. By making the test cases smaller, developers can understand, reproduce, and fix the issues more quickly and efficiently

Delta debugging is a commonly used method for test case reduction. It simplifies a test case step by step while keeping its ability to cause the issue intact. The method does this by dividing the input data into smaller parts and testing each one to see if it still causes the problem. If it does, the other parts are thrown out, and the process is repeated until the smallest possible input that still causes the issue is found