Project Report

SWE30009 Software Testing and Reliability

Student Name: Nguyen Nam Tung || **Student ID:** 103181157

Task 1: Present your understanding of the testing concepts

Subtask 1.1. Present your understanding of effectiveness metrics and how they can be applied to evaluate random testing and partition testing.

Effectiveness metrics can be defined as the metrics which are used to evaluate the performance of the testing strategies, particular how well they detect fault in the program. The most commonly used effectiveness metrics are P-measure, E-measure and F-measure.

- P-measure: This is the probability of detecting at least one failure
- E-measure: This is the expected probability of detecting failures
- F-measure: This is the expected number of test cases to detect the first failure

Let's compare and discuss the effectiveness of random testing and partition testing. A table will be provided to compare these 2 types of testing, given in testing a program which an input domain containing these 3 partitions:

- Partition 1: 100 inputs, 3 failure-causing inputs
- Partition 2: 200 inputs, 5 failure-causing inputs
- Partition 3: 250 inputs, 2 failure-causing inputs

We consider the number of test cases for this case is 100 test cases

Random Testing	Partition Testing
For random testing, tests case that are randomly selected from the entire input domain.	For partition testing, the input domain is divided into partitions and test cases are selected from each partition. The number of test cases are selected from each partition based on its size
Failure Ratio = $\frac{Number\ of\ failure{-}causing\ input}{Domain\ size}$ = $\frac{3+5+2}{100+200+250}$ = $\frac{1}{55}$ E measure = $\frac{1}{55}$ * 100 = 1.8181 P measure = 1 – (1 - $\frac{1}{55}$) ^{100} = 0.8404	Using proportional sampling, if we can allocate the number of tests to each partition: Partition 1 = 100 * $\frac{100}{550}$ = 18 test cases Partition 2 = 100 * $\frac{200}{550}$ = 36 test cases Partition 3 = 100 * $\frac{250}{550}$ = 46 test cases E measure = $\frac{3}{100}$ * 18 + $\frac{5}{200}$ * 36 + $\frac{2}{250}$ * 46 = 1.808 P measure = 1 – (1 - $\frac{3}{100}$)^{18} * (1 - $\frac{5}{200}$)^{36} * (1 - $\frac{2}{250}$)^{46} = 0.8395

Table 1: Random testing and partition testing comparison with 100 test cases

Next, let's consider the scenario where the number of test cases are only 10 test cases:

Random Testing	**Partition Testing**
Failure Ratio = $\frac{Number\ of\ failure-causing\ input}{Domain\ size}$ $= \frac{3+5+2}{100+200+250} = \frac{1}{55}$ E measure = $\frac{1}{55}$ * 10 = 0.1818 P measure = $1 - (1 - \frac{1}{55})^{100}$ = 0.1676	Using proportional sampling, if we can allocate the number of tests to each partition: Partition 1 = 10 * $\frac{100}{550}$ = 2 test cases Partition 2 = 10 * $\frac{200}{550}$ = 4 test cases Partition 3 = 10 * $\frac{250}{550}$ = 4 test cases E measure = $\frac{3}{100} * 2 + \frac{5}{200} * 4 + \frac{2}{250} * 4$ = 0.192 P measure = $1 - (1 - \frac{3}{100})^2 * (1 - \frac{5}{200})^4 * (1 - \frac{2}{250})^4$ = 0.1766

Table 2: Random testing and partition testing comparison with 10 test cases

From the tables and calculations, we can see an interesting trend between random testing and partition testing when considering different numbers of test cases. With a large number of test cases (100 test cases), random testing performed slightly better than partition testing. This can be seen in the higher E-measure (1.8181 for random testing compared to 1.808 for partition testing) and a higher P-measure (0.8404 for random testing compared to 0.8395 for partition testing). These values indicate that, over a large number of test cases, random testing is slightly more effective at detecting at least one failure and has a higher expected probability of detecting failures.

However, the scenario changes when the number of test cases is reduced. When we decreased the number of test cases from 100 to just 10, partition testing slightly outperformed random testing in both metrics. Partition testing had a higher E-measure (0.192 compared to 0.1818 for random testing) and a higher P-measure (0.1766 compared to 0.1676 for random testing). This suggests that partition testing is more effective with fewer test cases, which is likely due to its targeted nature, which divides the input domain into specific partitions and ensures that test cases are selected from every partition.

The proportional sampling strategy in partition testing is a simple method ensures that larger partitions receive more tests, based on their relative size. This strategy balances the testing effort across partitions, increasing the chance of detecting failures in partitions that are more heavily populated. All partitions must have the same sampling rate, which mean:

For $\sigma = \frac{Number\ of\ test\ cases}{Domain\ Size}$, then $\sigma_1 = \sigma_2 = \sigma_3 = \frac{18}{100} = \frac{36}{200} = \frac{46}{250}$

In the example above, Partition 3, which has a lower density of failure-causing inputs, receives the most tests (46 tests) because it is the largest partition. Partition 1, with a higher failure density, gets fewer tests (18 tests), but this is still proportional to its size.

Subtask 1.2. Present your understanding of metamorphic testing

Metamorphic testing is a software testing technique that focuses on addressing the problem of test oracles for untestable program.

Test oracle is a mechanism that verifies whether the output of a program is correct for a given input or not. It follows the principle of backward substitution and evaluation. For instance, we can take an example of the program that solve the equation $x^2 - 4 = 0$. If the program returns the result of + 2 and -2, we can do backward substitution of these result to the program to check whether if $(-2)^2 - 4 = 0$ and $2^2 - 4 = 0$. However, some system lacks test oracle or test oracle is too expensive to be utilized, leading to test oracle problem.

An untestable system is defined as the system in which the outputs of some inputs cannot be verified, typically due to the absence of test oracle. For example, consider a system designed to generate warnings and forecast natural disasters for a specific region. The outcomes of such a system, such as predicting the occurrence or severity of an earthquake, may not be directly verifiable until the event actually happens, making it challenging to evaluate the accuracy of the system's predictions in real time.

The motivation for metamorphic testing comes from the need to test systems where traditional testing methods are insufficient due to the lack of a clear test oracle as for testing. In complex systems it can be difficult or impossible to verify the correctness of outputs. The challenge becomes even more difficult in systems where the expected output for each input isn't known. In these cases, metamorphic testing proves to be incredibly useful.

The intuition behind metamorphic testing is while it may not be feasible to know the correct output for every individual input and sometime , it is often possible to predict how the output should change when the input is modified in specific ways. By focusing on metamorphic relations, we can determine whether a system behaves consistently under various transformations of the input, even without knowing the precise output.

Metamorphic relation is the foundation of metamorphic testing. It is defined as the relationship between the inputs and their corresponding outputs. Before diving deep into the example, some notation will be utilized:

- SI denote source inputs
- Fi denote follow-up inputs

- SO denote source outputs
- FO denote follow-up outputs

Example:

Let's take an example of a program that accept a list of positive integers to return the sum of all the elements in that list

MR1: The sum of the entire list should be the sum of the sums of two or more non-overlapping sub lists that partition the original list, so FO = SO

SI	FI	SO	FO
[3,2,1,4]	[3,2],[1,4]	10	10

Table 3: MR1

MR2: Add a constant value to each input should result in a predictable change in the output. Assume that the amount added is k and the number of elements in the lists is denoted as n, so SO + n*k = FO

SI	FI	SO	FO
[1,2,3,4]	[2,3,4,5]	10	14

Table 4: MR2

MR3: Multiply a constant value to each input should result in a predictable change in the output. Assume that the amount multiplied is k, so k*SO = FO

SI	FI	SO	FO
[1,2,3,4]	[2,4,6,8]	10	20

Table 5: MR3

The process of metamorphic testing comprises of the following steps:

1. Identify a metamorphic relation : Determine a relationship that should hold between inputs and outputs when modifications are applied to the inputs.
2. Define and execute source test cases: Create and run the initial test cases to establish source outputs.

3. Generate and execute follow-up test cases: Apply the identified MR to the source test cases to create and execute the follow-up test cases.
4. Verify the metamorphic relation: Compare the outputs of the source and follow-up test cases to ensure they align according to the MR.

There are a lot of applications of metamorphic testing. They can be used in search engines such as Google to ensure that the search engine behaves consistently when slight changes are made to the input. They can also be used in some weather forecasting system, to ensure the consistency of weather forecasting models.

Subtask 1.3. Present your understanding of the mutation testing.

Mutation testing can be defined as a software testing technique used to evaluate the effectiveness of test cases by introducing changes to a program's code, called mutants, and assessing whether the existing test cases can detect these changes.

A mutant is a version of the original program with a small change made to it. Each mutant is created by applying one mutation operator to the program. The goal of mutation testing is to ensure that the test suite is strong enough to kill mutants. There are two types of mutants, which are equivalent mutant, which are mutants that are equivalent to the original program and non-equivalent mutants, which are mutants that are not equivalent to the original program.

Mutation operators are predefined rules that introduce mutants to the code. These operators are designed to simulate common programming errors, such as modifying operators, variables,...

A mutant is considered killed when its output differs from that of the original program for the same test case. If the mutant survives, this indicates that the test suite might not be thorough enough to catch certain bugs. The effectiveness of a test suite is often evaluated using the mutation score, which is calculated using the following formula:

Mutation Score = $\frac{Number\ of\ killed\ mutants}{Total\ number\ of\ mutants}$

Below are five mutation operators and corresponding examples of 20 mutants generated by applying these operators:

1. Change of arithmetic operators

   Original: a = b + c

   Mutants:

   a = b - c

   a = b * c

   a = b / c

   a = b % c

2. Change of arithmetic variables

   Original: a = b + c

Mutants:

a = b + d

a = b + e

a = c + d

a = d + e

a = c + e

3. Change a variable to a constant

   Original: a = b + c

   Mutants:

   a = b + 1

   a = b + 2

   a = b + 3

4. Change of relational operators

   Original: if (a > b)

   Mutants:

   If (a >= b)

   If (a <= b)

   If (a != b)

   if (a < b)

   if (a == b)

5. Change a constant value

   Original: a = b + 1

   Mutants:

   a = b + 2

   a = b + 3

   a = b + 4

6. Change of array index

   Original: a = b[1]

   Mutants:

   a = b[2]

   a = b[3]

Task 2: Test a program of your choice

For the program of my choice for this assignment, I will use the program kth_largest_element.py and here is the link to the program:

https://github.com/TheAlgorithms/Python/blob/master/data_structures/arrays/kth_largest_element.py

The chosen program is an implementation for finding the kth largest element in a list. It comprises of two primary functions:

- partition: A helper function that partitions an array around a pivot element.
- kth_largest_element: Finds the kth largest element in an array using partitioning logic.

Below are the inputs and outputs of the program:

Args:

- arr: The list of numbers.
- position: The position of the desired kth largest element. For instance, if you want to return the second largest value in the list, position is 2.

Returns:

- int: The kth largest element.

Based on the program, I propose the following two metamorphic relations:

Metamorphic Relation 1 (MR1): Array Shuffling

The kth largest element should remain the same when the array is shuffled. Shuffling changes the order of elements, but the value of the kth largest element should not be affected.

Metamorphic Relation 2 (MR2): Element Removal

Removing elements smaller than the kth largest element should not affect the output. Removing elements larger than the kth largest should also produce the same result since they don't influence the kth largest value.

Based on the original program's code, we can generate over 30 mutants. Each mutant will be stored in the MUTANTS directory, following the naming format: kth_largest_element_mutant_n.py, where n represents the mutant number. To facilitate this process, I created a script called create_empty_mutants_files.py in the SUT directory, which generates the empty Python mutant files. After creating these files, I copied the original program into each one and then modified them according to the mutation table outlined below:

Mutant #	Line of Code	Original Code	Mutant Code
M1	34	pivot = arr[high]	pivot = arr[high – 1]
M2	35	I = low – 1	I = low
M3	37	if arr[j] >= pivot:	if arr[j-1] >= pivot:
M4	37	if arr[j] >= pivot:	if arr[j] < pivot:
M5	37	if arr[j] >= pivot:	if arr[j] >= 2:
M6	38	i += 1	I - = 1
M7	38	i += 1	i + = 2
M8	39	arr[i], arr[j] = arr[j], arr[i]	arr[i], arr[j] = arr[j – 2], arr[i]
M9	39	arr[I], arr[j] = arr[j], arr[i]	arr[i – 2], arr[j] = arr[j], arr[i]
M10	41	return I + 1	return i + 2

M11	41	return i + 1	return i – 1
M12	100	low, high = 0, len(arr) – 1	low, high = 1, len(arr) – 1
M13	101	while low <= high:	while low < high:
M14	101	while low <= high:	while low <= 8:
M15	102	if low > len(arr) – 1 or high < 0:	if low > len(arr) – 2 or high < 0:
M16	102	if low > len(arr) – 1 or high < 0:	if low > len(arr) – 1 and high < 0:
M17	103	return -1	return 0
M18	105	if pivot_index == position – 1:	if pivot_index > position – 1:
M19	105	if pivot_index == position – 1:	if pivot_index == position:
M20	106	return arr[pivot_index]	return arr[pivot_index – 1]
M21	107	elif pivot_index > position – 1:	elif pivot_index > position:
M22	107	elif pivot_index > position – 1:	elif pivot_index >= position – 1:
M23	108	high = pivot_index – 1	high = pivot_index – 2
M24	108	high = pivot_index – 1	high = 3
M25	110	low = pivot_index + 1	low = pivot_index + 2
M26	110	low = pivot_index + 1	low = 1
M27	111	return -1	return -2
M28	35	I = low – 1	i = 3
M29	41	return i + 1	return 1
M30	40	arr[I + 1], arr[high] = arr[high], arr[I + 1]	arr[I], arr[high] = arr[high], arr[I + 1]

Table 6: Mutants for the original program

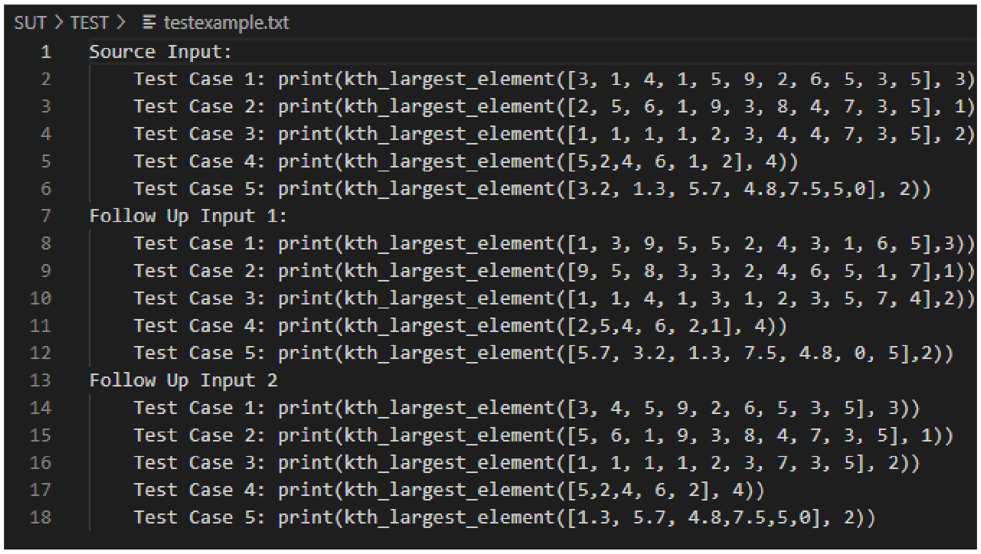
Five test cases have been selected for the testing process:

Test Case #	SI	FI-1 (MR1)	FI-2 (MR2)
1	([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5], 3)	([1, 3, 9, 5, 5, 2, 4, 3, 1, 6, 5],3) Modification: Shuffle the array	([3, 4, 5, 9, 2, 6, 5, 3, 5], 3) Modification: Remove 2 elements 1
2	([2, 5, 6, 1, 9, 3, 8, 4, 7, 3, 5], 1)	([9, 5, 8, 3, 3, 2, 4, 6, 5, 1, 7],1) Modification: Shuffle the array	([5, 6, 1, 9, 3, 8, 4, 7, 3, 5], 1) Modification: Remove the element 2
3	([1, 1, 1, 1, 2, 3, 4, 4, 7, 3, 5], 2)	([1, 1, 4, 1, 3, 1, 2, 3, 5, 7, 4],2) Modification: Shuffle the array	([1, 1, 1, 1, 2, 3, 7, 3, 5], 2) Modification: Remove 2 elements 4
4	([5,2,4, 6, 1, 2], 4)	([2,5,4, 6, 2,1], 4) Modification: Shuffle the array	([5,2,4, 6, 2], 4) Modification: Remove the element 1
5	([3.2, 1.3, 5.7, 4.8,7.5,5,0], 2)	([5.7, 3.2, 1.3, 7.5, 4.8, 0, 5],2) Modification: Shuffle the array	([1.3, 5.7, 4.8,7.5,5,0], 2) Modification: Remove the element 3.2

Table 7: Test Cases for each MR

To implement the tests, we can place the test cases in the main section of the program, with the results printed out to the terminal. A mutant is killed if the relationship between a test group and its outputs violates the MR. To ensure the MR is satisfied, we input the source input and follow-up Input into each mutant program to check whether the outputs align with the expected relation. If the outputs for both inputs fail to maintain the MR, the mutant killed.

The test cases, which are placed in the main section, are stored in text file in the TEST directory, following the format shown in Figure 1. For this assignment, I manually conducted all the testing by inputting the test cases into each mutant program.



SUT > TEST > testexample.txt

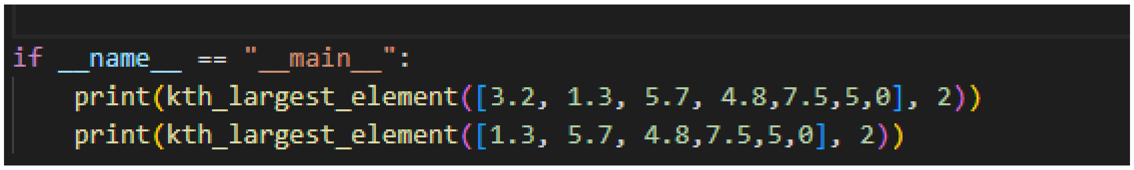
```
Source Input:
    Test Case 1: print(kth_largest_element([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5], 3)
    Test Case 2: print(kth_largest_element([2, 5, 6, 1, 9, 3, 8, 4, 7, 3, 5], 1)
    Test Case 3: print(kth_largest_element([1, 1, 1, 1, 2, 3, 4, 4, 7, 3, 5], 2)
    Test Case 4: print(kth_largest_element([5,2,4, 6, 1, 2], 4))
    Test Case 5: print(kth_largest_element([3.2, 1.3, 5.7, 4.8,7.5,5,0], 2))
Follow Up Input 1:
    Test Case 1: print(kth_largest_element([1, 3, 9, 5, 5, 2, 4, 3, 1, 6, 5],3))
    Test Case 2: print(kth_largest_element([9, 5, 8, 3, 3, 2, 4, 6, 5, 1, 7],1))
    Test Case 3: print(kth_largest_element([1, 1, 4, 1, 3, 1, 2, 3, 5, 7, 4],2))
    Test Case 4: print(kth_largest_element([2,5,4, 6, 2,1], 4))
    Test Case 5: print(kth_largest_element([5.7, 3.2, 1.3, 7.5, 4.8, 0, 5],2))
Follow Up Input 2
    Test Case 1: print(kth_largest_element([3, 4, 5, 9, 2, 6, 5, 3, 5], 3))
    Test Case 2: print(kth_largest_element([5, 6, 1, 9, 3, 8, 4, 7, 3, 5], 1))
    Test Case 3: print(kth_largest_element([1, 1, 1, 1, 2, 3, 7, 3, 5], 2))
    Test Case 4: print(kth_largest_element([5,2,4, 6, 2], 4))
    Test Case 5: print(kth_largest_element([1.3, 5.7, 4.8,7.5,5,0], 2))
```

Figure 1: Test cases

To run the test, for example with MR1, and to test mutants 1 through 30 using Test Case 1, the process is as follows:

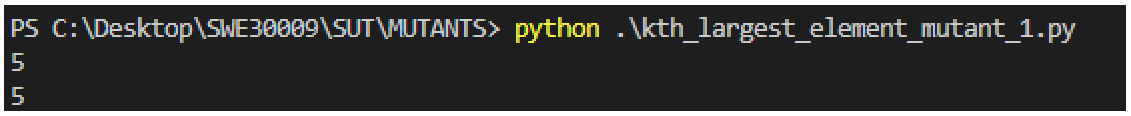
- Pick the source input from Test Case 1 and input it into each of the 30 mutants to generate the source output (SO).
- Next, pick the follow-up input corresponding to MR1 (for Test Case 1), and input it into each of the 30 mutants to generate the follow-up output (FO).
- For both MRs, the SO must be identical to the FO for the MR to be satisfied. If the SO is not equal to the FO, the mutant is considered killed.

This process is repeated for each test case, ensuring that the outputs align to determine if the mutant survives or is killed.



```
if __name__ == "__main__":
    print(kth_largest_element([3.2, 1.3, 5.7, 4.8,7.5,5,0], 2))
    print(kth_largest_element([1.3, 5.7, 4.8,7.5,5,0], 2))
```

Figure 2: Main section of the program



```
PS C:\Desktop\SWE30009\SUT\MUTANTS> python .\kth_largest_element_mutant_1.py
5
5
```

Figure 3: Sample output

MR 1: Array Shuffling (SO = FO)

Mutant #	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
M1	Not Killed	Not Killed	Killed	Not Killed	Killed
M2	Not Killed	Killed	Not Killed	Killed	Killed
M3	Killed	Killed	Not Killed	Killed	Killed
M4	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M5	Not Killed	Killed	Not Killed	Not Killed	Killed
M6	Killed	Killed	Killed	Killed	Killed
M7	Killed	Killed	Killed	Killed	Killed
M8	Not Killed	Killed	Killed	Not Killed	Not Killed
M9	Killed	Killed	Not Killed	Killed	Not Killed
M10	Killed	Killed	Killed	Killed	Killed
M11	Killed	Killed	Not Killed	Not Killed	Killed
M12	Not Killed	Not Killed	Not Killed	Killed	Not Killed
M13	Not Killed	Not Killed	Killed	Not Killed	Not Killed
M14	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M15	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M16	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M17	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M18	Not Killed	Killed	Killed	Killed	Killed
M19	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M20	Not Killed	Killed	Not Killed	Not Killed	Not Killed
M21	Not Killed	Not Killed	Killed	Not Killed	Not Killed
M22	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M23	Not Killed	Killed	Killed	Killed	Killed
M24	Killed	Killed	Killed	Not Killed	Killed

M25	Not Killed	Not Killed	Killed	Killed	Not Killed
M26	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M27	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M28	Killed	Killed	Killed	Killed	Killed
M29	Killed	Killed	Not Killed	Killed	Killed
M30	Not Killed	Killed	Killed	Killed	Killed

Table 8: Testing for MR1 with Test Suite 1

MR 2: Element Removal (SO = FO)

Mutant #	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
M1	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M2	Not Killed	Killed	Not Killed	Killed	Killed
M3	Not Killed	Killed	Not Killed	Not Killed	Not Killed
M4	Killed	Not Killed	Not Killed	Killed	Not Killed
M5	Not Killed	Killed	Not Killed	Not Killed	Killed
M6	Killed	Killed	Killed	Killed	Killed
M7	Killed	Killed	Not Killed	Killed	Killed
M8	Killed	Killed	Not Killed	Not Killed	Killed
M9	Killed	Killed	Not Killed	Killed	Killed
M10	Killed	Killed	Killed	Killed	Killed
M11	Not Killed	Not Killed	Killed	Killed	Killed
M12	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M13	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M14	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M15	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M16	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M17	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M18	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M19	Not Killed	Not Killed	Killed	Not Killed	Not Killed
M20	Not Killed	Killed	Not Killed	Not Killed	Not Killed
M21	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M22	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M23	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M24	Killed	Killed	Not Killed	Not Killed	Killed
M25	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M26	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M27	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed
M28	Killed	Killed	Killed	Killed	Killed
M29	Killed	Killed	Not Killed	Killed	Killed
M30	Not Killed	Not Killed	Not Killed	Not Killed	Not Killed

Table 9: Testing for MR2 with Test Suite 2

	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5	Test Suite	Average
MR1	9/30	16/30	13/30	13/30	14/30	21/30	0.4343
MR2	9/30	12/30	5/30	9/30	11/30	15/30	0.3067

Table 10: Mutation score result

Based on the table, we can analyse and compare the effectiveness of the two MRs across individual test cases and their overall mutation scores. MR1 with test suite 1 demonstrates a higher mutation score of 21/30 and an average mutation score of 0.4343 across all test cases, indicating that it is more effective at killing mutants. In contrast, MR2 with test suite 2 has a lower mutation score of 15/30 and a corresponding average mutation score of 0.3067.

	Test Case 1	Test Case 2	TC3	TC4	TC5
MR1	SO = FO	SO = FO	SO = FO	SO = FO	SO = FO
MR2	SO = FO	SO = FO	SO = FO	SO = FO	SO = FO

Table 11: Metamorphic Testing for both MRs for the 5 test cases for the original program

In conclusion, with both MRs successfully passing the metamorphic tests, as noted in Table 11, and given their high mutation scores, both MR1 and MR2 can be effectively used in the testing process. However, MR1 has demonstrated better performance, as reflected by its higher mutation score, making it the more effective MR for identifying faults and killing mutants in this particular testing scenario.