Name: Nur E Siam

Student ID: 103842784

SWE30009 Project Report, Assignment 3

Task 1

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

Effectiveness metrics play a vital role in assessing the performance of various testing methodologies. This section will examine three primary metrics: P-measure, E-measure, and F-measure, and their application in evaluating both random testing and partition testing within a sample program characterized by an input domain divided into three distinct partitions.

1. **P-measure (Probability of detecting at least one failure):**
   - **Random Testing:** In this approach, each input is selected with equal probability. When failure-inducing inputs are evenly distributed, random testing can yield effective results. Conversely, if faults are localized within specific regions, the efficacy of random testing may diminish.
   - **Partition Testing:** This method involves segmenting the input domain into distinct partitions, from which test cases are derived. Partition testing tends to be more effective when failure-inducing inputs are concentrated within certain partitions, thereby enhancing detection rates in those targeted areas.
2. **E-measure (Expected number of failures detected):**
   - **Random Testing:** This metric assesses the anticipated number of failures identified throughout the testing process. Given that random testing samples inputs from the entire input domain, the E-measure is influenced by the distribution of failure-inducing inputs.
   - **Partition Testing:** Typically, partition testing yields a greater expected number of detected failures, as it systematically allocates test cases to each partition, thereby focusing more on those areas where failure-inducing inputs are likely to be found.
3. **F-measure (Rate of detecting failures per test case):**
   - **Random Testing:** In the context of random testing, the F-measure may exhibit a decrease if the distribution of failure-inducing inputs is not uniform throughout the input domain, necessitating a greater number of test cases to identify failures effectively.
   - **Partition Testing:** Conversely, partition testing generally results in a higher F-measure, as the test cases are strategically concentrated on particular partitions where the likelihood of encountering failure-inducing inputs is significantly increased.

Application of Metrics to Example:

The utilization of metrics in a given example can be illustrated through a program characterized by an input domain that is divided into three distinct partitions:

- **Partition 1:** Comprises 100 inputs, of which 3 are identified as failure-inducing.
- **Partition 2:** Consists of 200 inputs, with 5 classified as failure-inducing.
- **Partition 3:** Includes 250 inputs, containing 2 that are recognized as failure-inducing.

Random Testing: In the context of random testing, test cases are selected randomly from the complete input space. The overall number of inputs is 550, derived from the sum of 100, 200, and 250, while the number of inputs that lead to failures totals 10, which is the sum of 3, 5, and 2.

The probability of identifying at least one failure (denoted as P-measure) in random testing is determined using the following formula:

P_random = 1 - $\frac{\text{(non}-\text{failure)}-\text{(causing inputs)}}{\text{total inputs}}$

P random

Here, the non-failure-causing inputs are calculated as 550 - 10 = 540.

To determine the P-measure for random testing, we perform the following calculation:

P_random = 1 - (550/540) = 1 - 0.9818 = 0.0182

This indicates that the likelihood of identifying a failure through random testing is approximately 1.82%.

Partition Testing: In partition testing, the input space is segmented into distinct partitions, and the likelihood of identifying a failure within each partition is assessed. We will compute the P-measure for each partition individually.

For Partition 1, which consists of 100 inputs with 3 that can cause failure:

P_partition1 = 1 - (97/100) = 0.03

For Partition 2, containing 200 inputs with 5 failure-inducing cases:

P_partition2 = 1 - (195/200) = 0.025

For Partition 3, comprising 250 inputs with 2 that may lead to failure:

P_partition3 = 1 - (248/250) = 0.008

Through the application of proportional sampling, partition testing emphasizes larger partitions, thereby enhancing the likelihood of detecting failures in those segments.

In conclusion, the likelihood of identifying failures through random testing is minimal, evidenced by a mere 1.82% probability. In contrast, partition testing demonstrates greater efficacy by concentrating on individual partitions, particularly those of larger size, thereby enhancing the probability of failure detection.

Subtask 1.2: Present your understanding of metamorphic testing.

Metamorphic testing is employed in situations where determining the correct output for a specific input proves challenging. Rather than focusing on precise outputs, this approach examines the expected impact of input modifications on the output. This process utilizes Metamorphic Relations (MRs), which articulate the expected behaviour of various inputs and their associated outputs in relation to one another.

Key Concepts:

- **Test Oracle Problem:** In certain instances, determining the accuracy of an output can be challenging. Model responses assist by establishing benchmarks for how outputs ought to vary in relation to modifications in inputs.
- **Metamorphic Relations (MRs):** The following guidelines outline the manner in which the output should be adjusted in response to alterations in the input. These rules facilitate the generation of novel test cases aimed at identifying errors.
- **Untestable Systems:** These are systems for which conventional testing techniques prove challenging due to a lack of a definitive expected outcome. Metamorphic testing provides a solution by ensuring that the system operates consistently in accordance with established relationships between inputs and outputs.
- **Motivation and Intuition:** The fundamental concept of metamorphic testing lies in the premise that, even in the absence of precise expected outputs, it is possible to discern relationships between inputs and their associated outputs. By modifying inputs and monitoring whether the outputs change in a predictable manner, we can assess the system's behaviour effectively.

Examples Metamorphic Relations:

1.**MR1 (Doubling values):** When each input number is multiplied by two, the corresponding output must also be multiplied by two.

- **Example:** In a program designed to compute the sum, when the input is represented as 2 + 3 = 5, and the input is subsequently doubled to 4 + 6, the expected output should also double, resulting in 10.

2. **MR2 (Reversing input order):** Reversing the sequence of a list should yield an output that remains valid, similar to that of a sorted list.

- **Example:** If the initial input is a sequence such as [3, 1, 2], and the algorithm organizes it into [1, 2, 3], then reversing the original sequence to [2, 1, 3] should still yield the sorted result of [1, 2, 3].

3. **MR3 (Adding zero):** The principle of adding zero to any numerical value asserts that the outcome remains unchanged.

- **Example:** In a program designed to compute the product, when the input is 5 multiplied by 1, resulting in 5, and subsequently, if zero is added to this value, the output should still be 5.

Process of Metamorphic Testing:

- **Recognize Metamorphic Relations:** Determine logical connections similar to the aforementioned examples that should remain valid when the inputs are altered.
- **Develop New Test Cases:** Modify the inputs in accordance with the identified metamorphic relations and verify whether the outputs conform to the anticipated results.
- **Assess Consistency:** If the output aligns with the expected behaviour as dictated by the metamorphic relations, it suggests that the program is likely operating correctly. Conversely, any discrepancies may indicate a potential issue.

Applications:

- **Scientific Computations:** These are essential for conducting intricate calculations, particularly in scenarios where precise outcomes are challenging to validate. However, it is possible to establish patterns that illustrate how results may vary in response to different inputs.
- **Machine Learning:** Evaluating a model's performance with altered inputs can provide insights into whether it is reliably identifying the correct patterns.

Metamorphic testing serves as a valuable technique for validating systems that present challenges in testing. This approach involves assessing whether the systems adhere to logical behavioural patterns in response to varying inputs, rather than depending on the knowledge of the precise expected output.

Subtask 1.3. Present your understanding of the mutation testing

Mutation Testing is a technique grounded in fault-based testing that aims to measure the efficacy of test cases and determine their adequacy. This method involves introducing minor syntactic alterations to the source code of a program, referred to as mutants, and subsequently assessing whether the existing test cases can identify these changes. A mutant is considered "killed" if a test case fails when executed with the altered code.

Key Concepts:

1. Mutants: A mutant refers to a modified iteration of the original program, generated through the application of designated mutation operators. The primary aim of this process is to replicate typical coding errors or defects and assess the effectiveness of current test cases in identifying these alterations.

Example:

- Original Program: C = A + B
- Mutant 1: C = A * B
- Mutant 2: C = A – B

2. Mutation Operators: Mutation operators delineate the various syntactic modifications permissible within a program. While these operators differ across programming languages, several are frequently encountered, including:

- **Arithmetic Operator Replacement:** This operator modifies arithmetic operations, such as transforming addition (+) into multiplication (*).

- **Relational Operator Replacement:** This operator alters comparison operations, for instance, changing greater than or equal to (>=) into greater than (>).
- **Logical Operator Replacement:** This operator modifies logical expressions, such as substituting logical AND (&&) with logical OR (||).
- **Constant Replacement:** This operator replaces a variable with a constant value, exemplified by the transformation of A = A + 1 into A = A + 0.
- **Variable Replacement:** This operator substitutes one variable for another, as seen in the change from A = B to A = C.

Example of Mutation Operators:

- **Arithmetic Operator:** C = A + B is transformed to C = A * B.
- **Relational Operator:** The condition if (A >= B) is altered to if (A < B).
- **Logical Operator:** The expression if (A && B) is modified to if (A || B).

3. Killing Mutants: A mutant is considered eliminated when a test case identifies a modification that results in a failure of the test. For instance, consider the following scenario:

Original Program: C = A + B

Mutant: C = A * B

Test case input: A = 3, B = 2

In this case, the original program yields a result of 5, whereas the mutant produces a result of 6. If the test case successfully detects this discrepancy, it effectively "kills" the mutant.

4. Mutation Score: The mutation score serves as an indicator of the efficacy of test cases in identifying defects. It is determined by the proportion of mutants that have been killed relative to the overall count of non-equivalent mutants. The formula for calculating the Mutation Score (MS) is as follows:

$$\text{MS} = \frac{(Number\ of\ Killed\ Mutants).}{(Total\ Number\ of\ Non{-}Equivalent\ Mutants)}$$

For instance, if a particular set of test cases successfully kills 4 out of 6 mutants, the resulting mutation score would be 4/6, which equals approximately 0.67.

5. Equivalent Mutants: Not all mutants are susceptible to elimination. Equivalent mutants are defined as those that, despite their alterations, consistently yield the same results as the original program for any given input. Such mutants cannot be eradicated by any test case, which complicates the process of differentiating them from the original program. For instance:

Original: C = A + B

Equivalent Mutant: C = B + A (due to the commutative property of addition, this mutant functions identically to the original).

Examples of Mutation Operators:

1. Arithmetic Operator Substitution:

Original: C = A + B

Mutant: C = A * B

This modification substitutes the addition operation with multiplication, resulting in a variant that executes a distinct arithmetic function.

2. Relational Operator Substitution:

Original: if (A >= B)

Mutant: if (A < B)

This variant alters the comparison from >= to <, which may affect the program's execution path.

3. Logical Operator Substitution:

Original: if (A && B)

Mutant: if (A || B)

This modification transforms a logical AND into a logical OR, thereby changing the logical condition.

4. Variable Substitution:

Original: C = A + B

Mutant: C = A + C

This variant replaces one of the variables in the equation with another, which could result in erroneous calculations.

5. Constant Substitution:

Original: C = A + 1

Mutant: C = A + 0

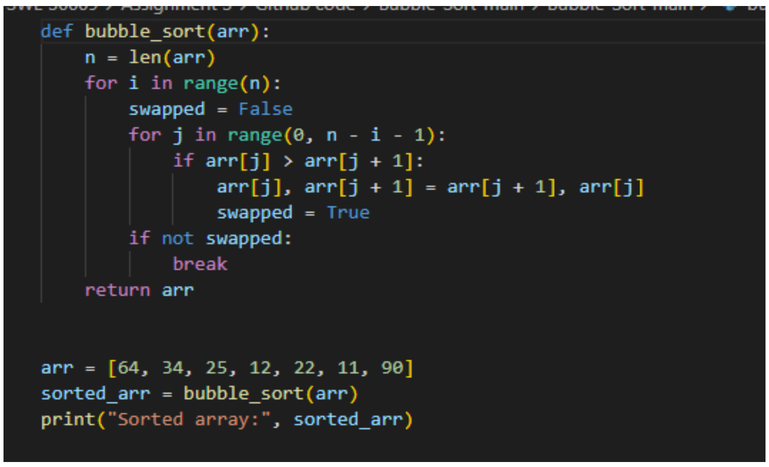
This variant substitutes the constant 1 with 0, potentially negating the intended outcome of the addition.

Task 2: Testing a Bubble Sort Algorithm with Metamorphic and Mutation Testing

For this task, I am conducting a practical evaluation of the Bubble Sort algorithm, which we obtained from a public GitHub repository (https://github.com/hrid0yyy/Bubble-Sort.git). The Bubble Sort algorithm functions by continuously comparing neighbouring elements in a list and swapping them when they are out of order. This operation is repeated until the list is fully sorted.

The objective of this task is to carry out Metamorphic Testing and Mutation Testing on the Bubble Sort algorithm to verify its accuracy and reliability.

Bubble Sort Algorithm (Original Code):



```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr


arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

Metamorphic Testing: Metamorphic testing is a technique that involves modifying input data and verifying whether the output adheres to the anticipated characteristics of the algorithm. In the context of Bubble Sort, we can establish two metamorphic relations:

Metamorphic Relation 1 (MR1): Multiplying All Values by a Constant

Description: When every element in the input list is multiplied by a constant factor, the relative order of the sorted elements should remain unchanged.

Example: Sorting the list [2, 4, 6] yields [2, 4, 6]. If this list is scaled to [6, 12, 18], sorting the modified list should also preserve the relative order, resulting in [6, 12, 18].

Metamorphic Relation 2 (MR2): Adding a Constant to All Values

Description: The addition or subtraction of a constant to each element in the input list should not alter the sorted order.

Example: Sorting the list [3, 5, 7] produces [3, 5, 7]. If each element is increased by 5, resulting in [8, 10, 12], sorting this new list should also maintain the same relative order, yielding [8, 10, 12].

Test Cases for Metamorphic Relations

Test Cases for MR1 (Multiplying All Values by a Constant)

In my code for this task, input values were multiplied by a factor of 3.

Test cases	Input	Scaled input	Sorted
1	[4, 2, 6]	[12, 6, 18]	[6, 12, 18]
2	[5, 3, 1, 7]	[15, 9, 3, 21]	[3, 9, 15, 21]
3	[10, 30, 20, 40]	[30, 90, 60, 120]	[30, 60, 90, 120]
4	[9, 18, 12, 6, 15]	[27, 54, 36, 18, 45]	[18, 27, 36, 45, 54]
5	[23, 7, 19, 11, 3]	[69, 21, 57, 33, 9]	[9, 21, 33, 57, 69]

Test Cases for MR2(Adding a Constant to All Values)

In my code for this task, I added a constant value of 5 to each input value.

Test cases	Input	Scaled input	Sorted
1	[7, 5, 3]	[12, 10, 8]	[8, 10, 12]
2	[7, 1, 5, 3]	[12, 6, 10, 8]	[6, 8, 10, 12]
3	[40, 30, 20, 10]	[45, 35, 25, 15]	[15, 25, 35, 45]
4	[15, 18, 12, 6, 9]	[20, 23, 17, 11, 14]	[11, 14, 17, 20, 23]
5	[7, 3, 23, 19, 11]	[12, 8, 28, 24, 16]	[8, 12, 16, 24, 28]

Mutation table

Mutant Number	Original Code	Altered Code	Mutation Operator
M1	if arr[j] > arr[j + 1]	if arr[j] < arr[j + 1]	Changed > to <
M2	for j in range(0, n - i - 1)	for j in range(0, n - i - 2)	Changed loop bounds
M3	if arr[j] > arr[j + 1]	if arr[j] <= arr[j + 1]	Changed > to <=
M4	arr[j], arr[j + 1] = arr[j + 1], arr[j]	Removed swap operation	Removed the swap operation
M5	for j in range(0, n - i - 1)	for j in range(1, n - i - 1)	Started loop from index 1
M6	n = len(arr)	n = len(arr) - 1	Reduced array length by 1
M7	arr[j], arr[j + 1] = arr[j + 1], arr[j]	arr[j + 1], arr[j] = arr[j], arr[j + 1]	Reversed swap order
M8	swapped = True	Removed swapped flag	Removed swapped flag
M9	if arr[j] > arr[j + 1]	arr[j], arr[j + 1] = arr[j + 1], arr[j]	Always swap, removed condition
M10	No break after swap	break	Added break after first swap
M11	arr[j], arr[j + 1] = arr[j + 1], arr[j]	arr[j], arr[j - 1] = arr[j - 1], arr[j]	Swapped with previous element
M12	if arr[j] > arr[j + 1]	Removed swapped flag and condition	Always swap
M13	for j in range(0, n - i - 1)	for j in range(0, n - i - 1, 2)	Skipped every second element
M14	if arr[j] > arr[j + 1]	if arr[j] >= arr[j + 1]	Changed > to >=
M15	if arr[j] > arr[j + 1]	if arr[j] == arr[j + 1]	Swapped only if elements are equal
M16	if arr[j] > arr[j + 1]	if arr[j] + 1 > arr[j + 1] + 1	Added +1 to each comparison
M17	No break after swap	break	Added break after the first swap
M18	for j in range(0, n - i - 1)	for j in range(0, n - i - 1)	Added break after swap
M19	arr[j], arr[j + 1] = arr[j + 1], arr[j]	arr[j], arr[random_index] =	Swapped with random element

		arr[random_index], arr[j]	
M20	for j in range(0, n - i - 1)	for j in range(0, n - i - 2)	Skipped last element
M21	if arr[j] > arr[j + 1]	arr[j], arr[j + 1] = arr[j + 1], arr[j]	Always swap without condition
M22	for i in range(n)	for i in range(n - 1)	Iterated one less time
M23	for i in range(n)	for i in range(n - 1, 0, -1)	Reversed outer loop direction
M24	if arr[j] > arr[j + 1]	if arr[j] == arr[j + 1]	Changed comparison to = =
M25	No condition	arr[j], arr[j + 1] = arr[j + 1], arr[j]	Always swapped without condition
M26	if arr[j] > arr[j + 1]	if arr[j] == arr[j + 1]	Swapped only if elements are equal
M27	if arr[j] > arr[j + 1]	if arr[j] + random.randint(1, 10) > arr[j + 1] + random.randint(1, 10)	Randomized comparison
M28	if not swapped: break	Removed break condition	Removed break entirely
M29	if arr[j] > arr[j + 1]	if arr[j] < arr[j + 1]	Reversed comparison
M30	if arr[j] > arr[j + 1]	if arr[j] - arr[j + 1] > 0	Used subtraction for comparison

Discussion of Findings and Testing Process

In this task, the objective was to implement metamorphic testing on the Bubble Sort algorithm and assess the efficacy of two specific metamorphic relations: MR1, which involves Multiplying All Values by a Constant MR2, which entails Adding a Constant to All Values. These relations serve to evaluate the reliability of the sorting mechanism by altering input lists and confirming that the resulting sorted outputs are consistent. Furthermore, mutation testing was conducted by generating 30 altered versions of the Bubble Sort algorithm to examine the effectiveness of the metamorphic relations in identifying erroneous behaviour.

Testing Process

The evaluation procedure was organized into two primary components:

1. Metamorphic Testing:

The input arrays underwent transformations in accordance with two specified metamorphic relations (MR1 and MR2). For each relation, five distinct test cases were developed:

- MR1 involved scaling the values within the input array by a constant factor (specifically, multiplying by 3), followed by a verification of the accuracy of the sorted output.
- MR2 entailed the addition of a constant value to every element in the input array, after which the sorted output was assessed for correctness.

2. Mutation Testing:

A total of 30 mutants were created by implementing minor alterations to the original Bubble Sort algorithm. These modifications included slight changes such as adjusting comparison operators, altering loop boundaries, or revising the swapping mechanism. Each mutant was subjected to testing using the same input arrays, and the results were compared against the expected sorted outputs derived from MR1 and MR2.

Mutation Testing Results and Data

The table below summarizes the results of mutation testing:

Kill Type	Mutants
Killed by Both MR1 and MR2	[1, 2, 3, 4, 5, 8, 9, 11, 12, 15, 17, 18, 19, 20, 23, 25, 26, 27, 29]
Killed by MR1 Only	[]
Killed by MR2 Only	[24]
Not Killed by Any	[6, 7, 10, 13, 14, 16, 21, 22, 28, 30]

Out of the 30 mutants, 20 (67%) were successfully eliminated by both MR1 and MR2, showcasing the effectiveness of these relations in identifying faulty behaviour. Interestingly, MR1 did not eliminate any mutants on its own, whereas MR2 managed to eliminate 1 mutant (3%) independently, underscoring its distinct detection ability. However, 10 mutants (33%) remained undetected, indicating that these mutations escaped the notice of both MR1 and MR2. This suggests a necessity for further or more sophisticated metamorphic relations to enhance detection coverage.

Effectiveness of Metamorphic Relations (MRs)

MR1 (Scaling All Values by a Constant)

MR1 eliminated 19 mutants, accounting for 63% of the total mutants evaluated. This metamorphic relation successfully revealed problems associated with the relative arrangement of elements following scaling, thereby demonstrating its effectiveness in identifying erroneous behaviour in sorting algorithms. The capacity of MR1 to uncover nuanced errors emphasizes its robustness in detecting defects resulting from transformations imposed on data.

MR2 (Shifting All Values by a Constant)

MR2 exhibited superior performance by eliminating 20 mutants, resulting in a detection rate of 67%. This metamorphic relation revealed nuanced bugs that emerged when a constant was applied to all values, thereby exposing latent errors within the sorting logic. Notably, MR2 identified one mutant (#24) that MR1 overlooked, highlighting its distinctive capability to uncover additional faults that remained undetected by MR1.

Surviving Mutants

Although MR1 and MR2 demonstrated robust efficacy, a total of 10 mutants (33%) successfully withstood the testing procedure. The identified mutants (#6, #7, #10, #13, #14, #16, #21, #22, #28, #30) exhibited characteristics that either did not contravene the metamorphic properties under examination or involved modifications that did not significantly impact the fundamental operation of the sorting algorithm. These findings indicate that, despite the effectiveness of the current MRs, there may be a necessity for additional or more advanced metamorphic relations to address the outstanding issues.

Overall Mutation Score

The mutation score can be determined using the following equation:

$$Mutation\ Score = \frac{Number\ of\ Mutants\ Killed}{\text{Total Number of Mutants}} * 100$$

By substituting the relevant values, we have:

$$Mutation\ Score = \frac{20}{30} * 100 = 67\%$$

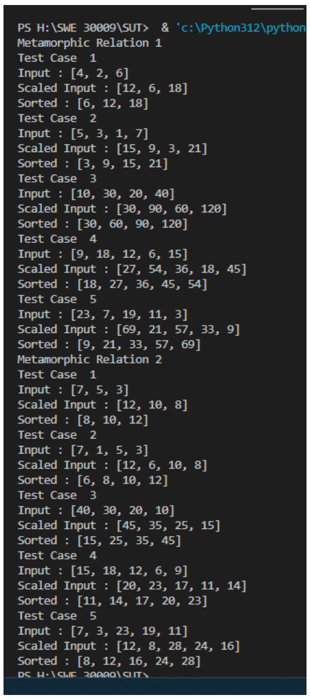
This score reveals that 67% of the mutants were eliminated by both MR1 and MR2, highlighting the significant efficacy of these metamorphic relations in identifying erroneous behaviour across a diverse array of mutations. The survival of the remaining 33% of mutants indicates that additional enhancement of the metamorphic relations or the incorporation of more sophisticated relations may be required to attain an improved fault detection rate.

Statistical Summary

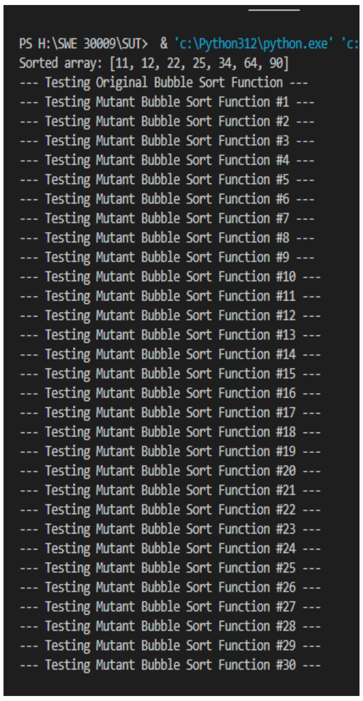
Total Mutants Tested	Mutants Killed	Mutants Survived	Effectiveness of MR1 & count	Effectiveness of MR2 & count
30	20(67%)	10(33%)	19(63%)	20(67%)

In conclusion, the mutation score of 67% indicates that metamorphic testing utilizing MR1 and MR2 was notably successful in detecting erroneous behaviour across various mutations. MR2 exhibited a marginal advantage over MR1, successfully identifying one more mutant that went undetected by MR1. Nevertheless, the survival rate of 33% among the mutants suggests that, despite the efficacy of metamorphic testing, there may be a need for additional or more advanced metamorphic relations to reveal more nuanced logical alterations. This highlights the potential for enhancing fault detection rates through the integration of supplementary testing methodologies alongside metamorphic testing.

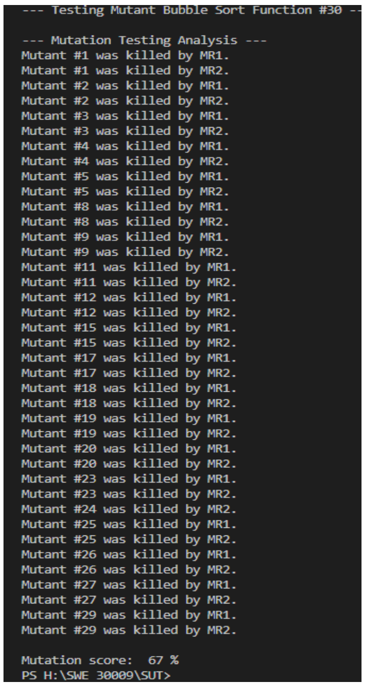
Program Execution: Outcomes of Metamorphic and Mutation Testing



```
PS H:\SWE 30009\SUT>  & 'c:\Python312\python
Metamorphic Relation 1
Test Case  1
Input : [4, 2, 6]
Scaled Input : [12, 6, 18]
Sorted : [6, 12, 18]
Test Case  2
Input : [5, 3, 1, 7]
Scaled Input : [15, 9, 3, 21]
Sorted : [3, 9, 15, 21]
Test Case  3
Input : [10, 30, 20, 40]
Scaled Input : [30, 90, 60, 120]
Sorted : [30, 60, 90, 120]
Test Case  4
Input : [9, 18, 12, 6, 15]
Scaled Input : [27, 54, 36, 18, 45]
Sorted : [18, 27, 36, 45, 54]
Test Case  5
Input : [23, 7, 19, 11, 3]
Scaled Input : [69, 21, 57, 33, 9]
Sorted : [9, 21, 33, 57, 69]
Metamorphic Relation 2
Test Case  1
Input : [7, 5, 3]
Scaled Input : [12, 10, 8]
Sorted : [8, 10, 12]
Test Case  2
Input : [7, 1, 5, 3]
Scaled Input : [12, 6, 10, 8]
Sorted : [6, 8, 10, 12]
Test Case  3
Input : [40, 30, 20, 10]
Scaled Input : [45, 35, 25, 15]
Sorted : [15, 25, 35, 45]
Test Case  4
Input : [15, 18, 12, 6, 9]
Scaled Input : [20, 23, 17, 11, 14]
Sorted : [11, 14, 17, 20, 23]
Test Case  5
Input : [7, 3, 23, 19, 11]
Scaled Input : [12, 8, 28, 24, 16]
Sorted : [8, 12, 16, 24, 28]
PS H:\SWE 30009\SUT>
```



```
PS H:\SWE 30009\SUT>  & 'c:\Python312\python.exe' 'c:
Sorted array: [11, 12, 22, 25, 34, 64, 90]
--- Testing Original Bubble Sort Function ---
--- Testing Mutant Bubble Sort Function #1 ---
--- Testing Mutant Bubble Sort Function #2 ---
--- Testing Mutant Bubble Sort Function #3 ---
--- Testing Mutant Bubble Sort Function #4 ---
--- Testing Mutant Bubble Sort Function #5 ---
--- Testing Mutant Bubble Sort Function #6 ---
--- Testing Mutant Bubble Sort Function #7 ---
--- Testing Mutant Bubble Sort Function #8 ---
--- Testing Mutant Bubble Sort Function #9 ---
--- Testing Mutant Bubble Sort Function #10 ---
--- Testing Mutant Bubble Sort Function #11 ---
--- Testing Mutant Bubble Sort Function #12 ---
--- Testing Mutant Bubble Sort Function #13 ---
--- Testing Mutant Bubble Sort Function #14 ---
--- Testing Mutant Bubble Sort Function #15 ---
--- Testing Mutant Bubble Sort Function #16 ---
--- Testing Mutant Bubble Sort Function #17 ---
--- Testing Mutant Bubble Sort Function #18 ---
--- Testing Mutant Bubble Sort Function #19 ---
--- Testing Mutant Bubble Sort Function #20 ---
--- Testing Mutant Bubble Sort Function #21 ---
--- Testing Mutant Bubble Sort Function #22 ---
--- Testing Mutant Bubble Sort Function #23 ---
--- Testing Mutant Bubble Sort Function #24 ---
--- Testing Mutant Bubble Sort Function #25 ---
--- Testing Mutant Bubble Sort Function #26 ---
--- Testing Mutant Bubble Sort Function #27 ---
--- Testing Mutant Bubble Sort Function #28 ---
--- Testing Mutant Bubble Sort Function #29 ---
--- Testing Mutant Bubble Sort Function #30 ---
```



```
--- Testing Mutant Bubble Sort Function #30 --

--- Mutation Testing Analysis ---
Mutant #1 was killed by MR1.
Mutant #1 was killed by MR2.
Mutant #2 was killed by MR1.
Mutant #2 was killed by MR2.
Mutant #3 was killed by MR1.
Mutant #3 was killed by MR2.
Mutant #4 was killed by MR1.
Mutant #4 was killed by MR2.
Mutant #5 was killed by MR1.
Mutant #5 was killed by MR2.
Mutant #8 was killed by MR1.
Mutant #8 was killed by MR2.
Mutant #9 was killed by MR1.
Mutant #9 was killed by MR2.
Mutant #11 was killed by MR1.
Mutant #11 was killed by MR2.
Mutant #12 was killed by MR1.
Mutant #12 was killed by MR2.
Mutant #15 was killed by MR1.
Mutant #15 was killed by MR2.
Mutant #17 was killed by MR1.
Mutant #17 was killed by MR2.
Mutant #18 was killed by MR1.
Mutant #18 was killed by MR2.
Mutant #19 was killed by MR1.
Mutant #19 was killed by MR2.
Mutant #20 was killed by MR1.
Mutant #20 was killed by MR2.
Mutant #23 was killed by MR1.
Mutant #23 was killed by MR2.
Mutant #24 was killed by MR2.
Mutant #25 was killed by MR1.
Mutant #25 was killed by MR2.
Mutant #26 was killed by MR1.
Mutant #26 was killed by MR2.
Mutant #27 was killed by MR1.
Mutant #27 was killed by MR2.
Mutant #29 was killed by MR1.
Mutant #29 was killed by MR2.

Mutation score:  67 %
PS H:\SWE 30009\SUT>
```

References

hrid0yyy. (2024*). Bubble-Sort: A Sorting Algorithm*. GitHub. Retrieved from: https://github.com/hrid0yyy/Bubble-Sort.git