Name: Dang Khoa Le – ID: 103844421

**Project Report**

**Task 1: Present your understanding of testing concepts**

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

**A. Effectiveness Metrics**

Effectiveness metrics are used to evaluate how well a testing strategy detects faults. There are three primary metrics often used to assess testing effectiveness:

1. **P-measure (Probability of Detecting at Least One Failure):**

P-measure calculates the probability that the testing strategy will detect at least one failure during the test. It assesses how likely the test is to uncover a fault, given that some failure-causing inputs exist in the input domain. A higher P-measure indicates a greater chance of detecting at least one failure.

Formula:

Random Testing

Pr ​= 1 − (1 − θ)n

Where:

θ = m/d: failure rate

n: number of test cases

m: number of failure-causing inputs

d: total size of the input domain.

Partition Testing

P​p = 1 -

Where:

k: number of partitions,

θi​ =​​ mi/di: failure rate in partition i

ni: number of tests allocated to partition i

mi: number of failure-causing inputs in i

di: domain size of partition i.

​

1. **E-measure (Expected Number of Failures):**

E-measure represents the expected number of failures that will be detected in the testing process. It measures the effectiveness of a testing strategy by estimating how many failure-causing inputs can be found during testing. The higher the E-measure, the more failures we expect to detect.

Formula:

Random Testing

Er​ = n\*θ

Where:

θ = m/d: failure rate

n: number of test cases

Partition Testing

E​p =

Where:

k: number of partitions,

θi​ =​​ mi/di: failure rate in partition i

ni: number of tests allocated to partition i

mi: number of failure-causing inputs in i

di: domain size of partition i.

1. **F-measure (Expected Number of Test Cases to Detect the First Failure):**

F-measure measures the number of test cases that need to be executed before the first failure is detected. A lower F-measure indicates that the testing strategy is efficient in finding a failure early.

Formula:

Random Testing

Fr​ = ​ = ​

Where:

θ = m/d: failure rate,

d: total size of the input domain,

m: number of failure-causing inputs.

Partition Testing

Fp =

Where:

k: number of partitions,

θi​ =​​ mi/di: failure rate in partition i

ni: number of tests allocated to partition

**Random Testing and Partition Testing**

* **Random Testing:**
* Random Testing involves selecting test cases randomly and independently from the input domain without considering specific partitions.
* The method tends to be less efficient in detecting faults because it lacks focus on critical areas of the input domain where failure-causing inputs may exist.
* **Partition Testing:**
* Partition testing divides the input domain into partitions (or sub-domains) based on certain criteria, such as equivalence classes. Each partition is then tested.
* This method can be more effective than random testing as it ensures focused testing on different partitions, which may contain failure-causing inputs.

**B. Program:**

Consider a program with an input domain divided into **3 partitions**:

* Partition 1: size = 100, contains 3 failure-causing inputs.
* Partition 2: size = 200, contains 5 failure-causing inputs.
* Partition 3: size = 250, contains 2 failure-causing inputs.

**I. Random Testing:**

1. **P-measure:**
   * Total domain size d = 100 + 200 + 250 = 550
   * Total failure-causing inputs m = 3 + 5 + 2 = 10
   * Failure rate θ = ​ ≈ 0.01818

If we run 20 tests (n = 20):

Pr ​= 1 − (1 − θ)n = 1 − (1 – 0.01818)20 = 1 − (0.98182)20 = 1−0.6931 = 0.3069

There is a **30.69%** probability of detecting at least one failure after 20 tests in random testing.

1. **E-measure:**

Er​ = n\*θ = 20\*0.01818 = 0.3636

We expect to detect approximately **0.364 failures** after 20 random tests.

1. **F-measure:**

Fr​ = ​ = ​ = = 55

On average, **55 tests** are needed to detect the first failure using random testing.

**II. Partition Testing:**

Proportional Sampling Strategy (PSS): To allocate the tests proportionally across the partitions based on their sizes, we scale the number of tests by the ratio of each partition size (100, 200, and 250) to the total domain size (550), and 20 tests are proportionally allocated as follows:

 Partition 1 (size 100) gets \* 20 ≈ 4 tests.

 Partition 2 (size 200) gets \* 20 ≈ 7 tests.

 Partition 3 (size 250) gets \* 20 ≈ 9 tests.

Using this approach, we distribute the 20 available test cases as follows: 4 tests for Partition 1, 7 tests for Partition 2, and 9 tests for Partition 3. This allocation allows us to maintain the same total number of test cases as random testing, facilitating a direct comparison between random and partition testing.

1. **P-measure:**
   * For Partition 1: θ1 = =
   * For Partition 2: θ2 = =
   * For Partition 3: θ3 = =

Pp​ = 1 - = 1 − (1 − 0.03)4 \* (1 − 0.025)7 \* (1 − 0.008)9

Pp ≈ 1 − 0.8853 \* 0.8376 \* 0.9303 = 0.3102

There is a **31.02%** probability of detecting at least one failure in partition testing.

1. **E-measure:**

Ep ​= = 4 \* 0.03 + 7 \* 0.025 + 9 \* 0.008 = 0.12 + 0.175 + 0.072 = 0.367

We expect to detect approximately **0.367 failures** after these tests in partition testing.

1. **F-measure**

For each partition, we calculate the individual contributions to the F-measure:

Partition 1: F1 = = ≈ 35

Partition 2: F2 = = ≈ 43

Partition 3: F3 = = ≈ 129

On average, **35 tests** are needed to detect the first failure in Partition 1, **43 tests** are needed to detect the first failure in Partition 1, **129 tests** are needed to detect the first failure in Partition 1, using partition testing.

By which the overall **F-measure** for partition testing is:

Fp = = 35 + 43 + 129 = 207

Which indicates that it would take an average of 207 test cases to detect the first failure across all partitions.

**III. Effectiveness of Random Testing vs. Partition Testing:**

* + **P-measure**: Partition testing provides a slightly better probability of detecting at least one failure due to the more focused testing strategy.
  + **E-measure**: Both methods perform similarly in terms of the expected number of failures detected, with partition testing having a minor advantage.
  + **F-measure**: Random testing is much more effective in detecting the first failure compared to partition testing, which requires more tests due to the failure-causing inputs being spread across different partitions.

In conclusion, **partition testing** offers a slight advantage in terms of overall failure detection (P-measure and E-measure), but **random testing** is more efficient at detecting the first failure (F-measure), making it more considerable in scenarios where detecting early failures is critical.

**Subtask 1.2. Present your understanding of metamorphic testing**

**1. Test Oracle**

A **test oracle** is a mechanism or principle that determines whether the output of a program is correct for a given input. However, in some cases, it is difficult to establish what the correct output should be, leading to the concept of **untestable systems**.

**2. Untestable Systems**

An **untestable system** refers to a program is said to be untestable or non-testable if the outputs of some inputs cannot be verified. This could be due to:

* The complexity of the problem space,
* Non-deterministic behaviour of the program,
* Lack of a reference model or expected output.

Examples of untestable systems include simulations, large-scale numerical computations, machine learning models, embedded system, and natural language processing (NLP) system.

In these cases, conventional testing methods that rely on a clear test oracle may not be applicable. This leads us to **metamorphic testing** as a solution.

**3. The Motivation and Intuition**

**Motivation:** Metamorphic Testing (MT) is motivated by the need to test programs where:

* Program is difficult or unavailable to define a complete or accurate test oracle.
* Increase testing coverage.
* Automate testing process for complex problems.

In these scenarios, instead of checking whether the output is correct for a specific input, we define **metamorphic relations (MRs)** between multiple inputs and outputs, and use these relations to validate the program.

**Intuition:** The key intuition behind metamorphic testing is instead of relying on knowing the exact correct output for each input, we can leverage the relationships between different inputs and their corresponding outputs. By applying transformations to inputs, we can check whether the expected behaviour between inputs and outputs (defined by the MRs) is maintained.

**4. Metamorphic Relations (MRs)**

A **metamorphic relation (MR)** describes how the output of a program should change in response to a specific transformation of the input. If the program behaves consistently with the MR after the input transformation, we assume that the program is behaving correctly.

**5. Process of Metamorphic Testing**

The process of metamorphic testing can be broken down into the following steps:

* Identify a source set of test cases.
* Apply metamorphic transformation to the input.
* Run the program on the transformed input (followed-up test case).
* A diagram of a process

  Description automatically generatedCompare the new output with the expected outcome derived from the MR. If the MR is violated, then the program has a fault (fail), otherwise, it passes.

Figure. Illustration on metamorphic testing process

**6. Applications of Metamorphic Testing**

Metamorphic testing can be applied to:

* **Machine learning models**: Testing non-deterministic models by checking if certain input-output relationships hold, such as increasing the value of a feature leading to a corresponding increase or decrease in the prediction.
* **Scientific simulations**: Validating simulations by transforming initial conditions and observing how the results change in predictable ways.
* **Search engines**: Testing search algorithms by modifying query structures and checking if the results remain logically consistent.
* **Numerical software**: Testing numerical computations by applying transformations to inputs that should produce predictable changes in the outputs.

**7. Examples of Metamorphic Relations (MRs)**

These followings show 3 different example programs as how they are applied with 3 Metamorphic Relations (MRs).

**Example 1: Sorting Algorithm:** Consider a program that provides a list with sorting algorithms (smallest to largest), which also removes duplicates and allows adding elements to its list.

**MR1: Random Reordering**

* **Source test case(s)**
  + Input(s): [2, 4, 6, 8, 10]
  + Output(s): [2, 4, 6, 8, 10]
* **Followed-up test case(s)**
  + Input(s): [4, 8, 2, 6, 10]
  + Output(s): [2, 4, 6, 8, 10]
* **Verify MR**: [2, 4, 6, 8, 10] == [2, 4, 6, 8, 10] (expected output is the same)

**MR2: Input with Duplicates (duplicates multiple elements)**

* **Source test case(s)**
  + Input(s): [2, 4, 6, 8, 10]
  + Output(s): [2, 4, 6, 8, 10]
* **Followed-up test case(s)**
  + Input(s): [2, 4, 6, 8, 10, 2, 6, 10]
  + Output(s): [2, 4, 6, 8, 10]
* **Verify MR**: [2, 4, 6, 8, 10] == [2, 4, 6, 8, 10] (duplicates are removed)

**MR3: Input with Negatives (adding constant -4)**

* **Source test case(s)**
  + Input(s): [2, 4, 6, 8, 10]
  + Output(s): [2, 4, 6, 8, 10]
* **Followed-up test case(s)**
  + Input(s): [2, 4, 6, 8, 10, -4]
  + Output(s): [-4, 2, 4, 6, 8, 10]
* **Verify MR**: [2, 4, 6, 8, 10] =/= [-4, 2, 4, 6, 8, 10] (expected output changes due to the negative number)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Source Test Case | Followed-up Test Case | Source Output | Followed-up Output |
| MR1 | [2, 4, 6, 8, 10] | [4, 8, 2, 6, 10] | [2, 4, 6, 8, 10] | [2, 4, 6, 8, 10] |
| MR2 | [2, 4, 6, 8, 10] | [2, 4, 6, 8, 10, 2, 6, 10] | [2, 4, 6, 8, 10] | [2, 4, 6, 8, 10] |
| MR3 | [2, 4, 6, 8, 10] | [2, 4, 6, 8, 10, -4] | [2, 4, 6, 8, 10] | [-4, 2, 4, 6, 8, 10] |

**Example 2: Numerical Computation:** Consider a program that sum up its list and allows adding elements to the list.

**MR1: Random Reordering**

**Source test case(s)**

* + Input(s): [7, 10, 1]
  + Output(s): 18
* **Followed-up test case(s)**
  + Input(s): [1, 7, 10]
  + Output(s): 18
* **Verify MR**: 18 == 18 (reordering wouldn’t change the sum value)

**MR2: Input with Duplicates (duplicates multiple elements)**

* **Source test case(s)**
  + Input(s): [7, 10, 1]
  + Output(s): 18
* **Followed-up test case(s)**
  + Input(s): [7, 10, 1, 10]
  + Output(s): 28
* **Verify MR**: 18 =/= 28 (duplication change the sum value)

**MR3: Input with Negatives (adding constant -4)**

* **Source test case(s)**
  + Input(s): [7, 10, 1]
  + Output(s): 18
* **Followed-up test case(s)**
  + Input(s): [7, 10, 1, -4]
  + Output(s): 14
* **Verify MR**: 18 =/= 14 (the sum decreases due to the negative input)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Source Test Case | Followed-up Test Case | Source Output | Followed-up Output |
| MR1 | [7, 10, 1] | [1, 7, 10] | 18 | 18 |
| MR2 | [7, 10, 1] | [7, 10, 1, 10] | 18 | 28 |
| MR3 | [7, 10, 1] | [7, 10, 1, -4] | 18 | 14 |

**Example 3: Numerical Multiplication:** Consider a program that multiply its list and allows adding elements to the list.

**MR1: Random Reordering**

* **Source test case(s)**
  + Input(s): [2, 3, 4, 5]
  + Output(s): 120
* **Followed-up test case(s)**
  + Input(s): [3, 5, 4, 2]
  + Output(s): 120
* **Verify MR**: 120 == 120 (reordering wouldn’t change the product of the list)

**MR2: Input with Duplicates (duplicates multiple elements)**

* **Source test case(s)**
  + Input(s): [2, 3, 4, 5]
  + Output(s): 120
* **Followed-up test case(s)**
  + Input(s): [2, 3, 4, 5, 2, 4]
  + Output(s): 960
* **Verify MR**: 120 =/= 960 (duplication change the product)

**MR3: Input with Negatives (adding constant -4)**

* **Source test case(s)**
  + Input(s): [2, 3, 4, 5]
  + Output(s): 120
* **Followed-up test case(s)**
  + Input(s): [2, 3, 4, 5, -4]
  + Output(s): -480
* **Verify MR**: 120 =/= -480 (output changes due to multiplication by a negative number)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Source Test Case | Followed-up Test Case | Source Output | Followed-up Output |
| MR1 | [2, 3, 4, 5] | [3, 5, 4, 2] | 120 | 120 |
| MR2 | [2, 3, 4, 5] | [2, 3, 4, 5, 2, 4] | 120 | 960 |
| MR3 | [2, 3, 4, 5] | [2, 3, 4, 5, -4] | 120 | -480 |

**Subtask 1.3. Present your understanding of the mutation testing.**

**Mutation Testing Overview**

Mutation testing is a type of white-box testing technique that assesses the quality of a test suite by introducing small changes (mutants) into a program and evaluating if the test suite can detect these changes. The goal is to ensure that the test cases are effective at catching errors by simulating real faults.

 **Mutants**: A **mutant** is a slightly altered version of the original program, generated by applying mutation operators. Each mutant represents a possible mistake a developer might make.

 **Mutation Operators**: Mutation operators are systematic rules used to generate mutants. These operators simulate common programming mistakes, such as replacing arithmetic or logical operators, modifying variables, or changing constants.

* **Killing Mutants**

A mutant is said to be **killed** when the output of the mutated program differs from the output of the original program for the same test case. The goal is to design test cases that are able to kill as many mutants as possible, ensuring that the test suite is robust enough to detect errors.

Example:

* Original Program: a = b + c
* Mutant (Arithmetic Operator Replacement): a = b - c
  + Test Case: b = 5, c = 3
  + Original Output: a = 8
  + Mutant Output: a = 2
  + The test case kills the mutant because the output differs from the original.
* **Mutation Score**

The **mutation score** is a measure of the effectiveness of a test suite. It is calculated using the following formula:

Mutation Score (MS) =

With:

k: the number of killed mutants

m: the total number of (non-equivalent) mutants

A higher mutation score indicates that the test suite is more effective in identifying errors.

Example:

* If there are 20 mutants generated, and 16 are killed, the mutation score would be:

MS = = 0.8

* This means the test suite was able to detect 80% of the injected faults (mutants).

**Test Case Example:**

**Mutation Score for Each Test Case:**

* **Test Case 1**: MS1 = = = 0.6
* **Test Case 2**: MS2 = = = 0.8
* **Test Case 3**: MS3 = = = 1.0
* **Test Case 4**: MS4 = = = 0.6
* **Test Case 5**: MS5 = = = 1.0
* **Test Case 6**: MS6 = = = 1.0

**Average Mutation Score:**

MSaverage = = = 0.83

The average mutation score across all test cases is **0.83**.

**Mutation Score for the Test Suite:**

Now, we check if any of the mutants were killed by any test case:

* **M1**: Killed by Test Cases 2, 3, 5, and 6.
* **M2**: Killed by Test Cases 1, 2, 3, 4, 5, and 6.
* **M3**: Killed by Test Case 3, 4, 5, and 6.
* **M4**: Killed by Test Cases 1, 2, 3, 4, 5, and 6.
* **M5**: Killed by Test Cases 1, 2, 3, 5, and 6.

MStest\_suite = = = 1.0

The mutation score for the entire test suite is **1.0**, meaning all mutants are killed by at least one test case.

**Conclusion:**

* The **average mutation score** across all test cases is **0.83**.
* The **mutation score for the test suite** is **1.0**, indicating the test suite is effective in killing all mutants.
* By using various mutation operators and crafting targeted test cases, we ensured that the majority of the mutants were killed, which demonstrates the effectiveness of the mutation testing process.

**Task 2: Test a program of your choice**

**1. Introduction**

Program: Matrix Multiplication Recursion

Source:<https://github.com/TheAlgorithms/Python/blob/master/matrix/matrix_multiplication_recursion.py>

Language: Python

In task 2, we apply **metamorphic relation testing** and **mutation testing** to a recursive matrix multiplication algorithm sourced from public repository “Matrix Multiplication Recursion”. The goal is to evaluate the program’s correctness and fault tolerance by observing how mutations affect its behaviour and whether the metamorphic relations defined for the program detect these faults.

This program was chosen due to its recursive nature and suitability for metamorphic relations, which consists of multiple types of operator, allowing more advantages on applying different mutation operators. The program’s complexity is sufficiently balanced, not too trivial, and not excessively complex.

**2. Proposed Metamorphic Relations**

**MR1: Scaling the Matrix**

* Definition: If both matrices (A and B) are scaled by a constant factor (2), the result of the multiplication should also be scaled by the same factor.
* Example: For A = [[1, 2], [3, 4]] and B = [[5, 6], [7, 8]], scaling both by 2 results in new matrices A' and B'. The product of A' \* B' should be twice the original product of A \* B.
* Intuition: Matrix multiplication maintains proportionality, so if the elements of both matrices are multiplied by a constant, the resulting matrix should reflect that scaling.

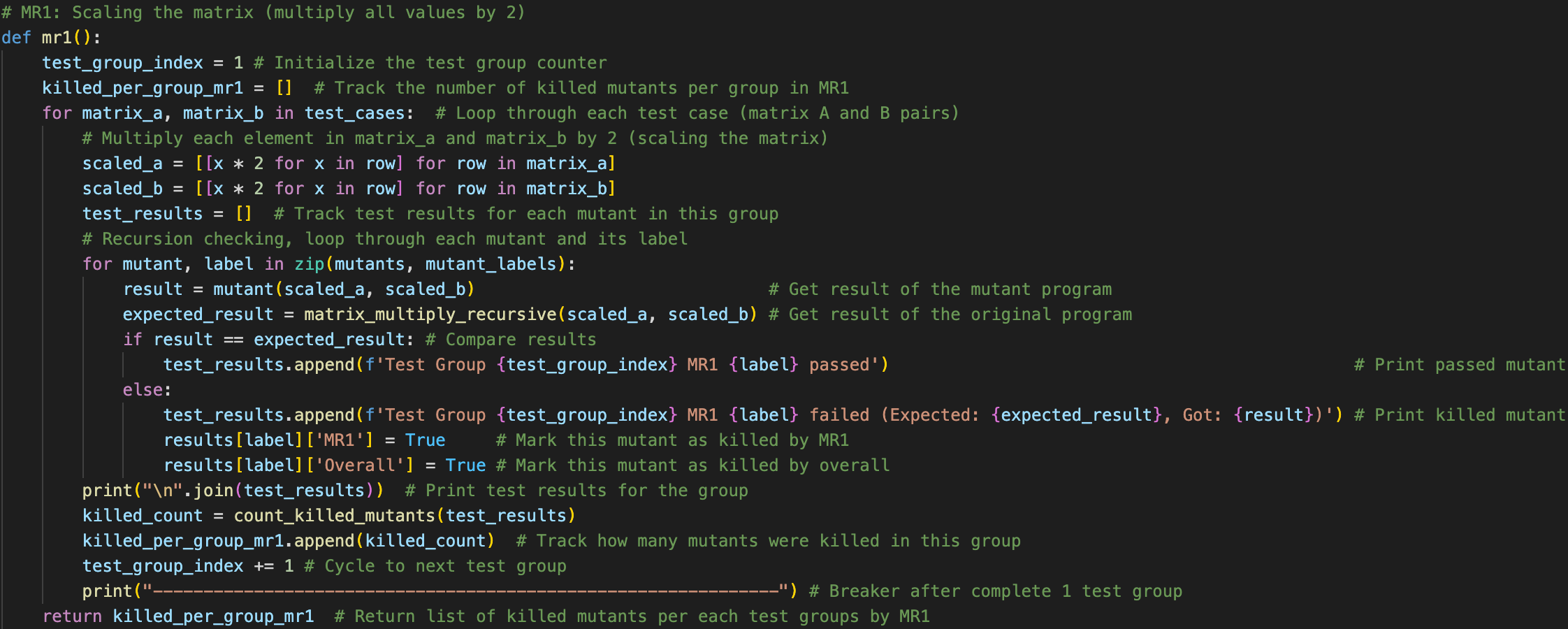
**MR2: Zeroing Last Rows and Columns**

* Definition: If the last row of matrix A and the last column of matrix B are replaced by zeros, the corresponding rows and columns of the result matrix should be zeros.
* Example: If A = [[1, 2], [3, 4]] and B = [[5, 6], [7, 8]], modifying A and B as A' = [[1, 2], [0, 0]] and B' = [[5, 0], [7, 0]], then the resulting product matrix should have zero entries in the last row and column.
* Intuition: If the last row or column contains only zeros, the resulting matrix should have zero entries in the corresponding places.

**3. Implementations**

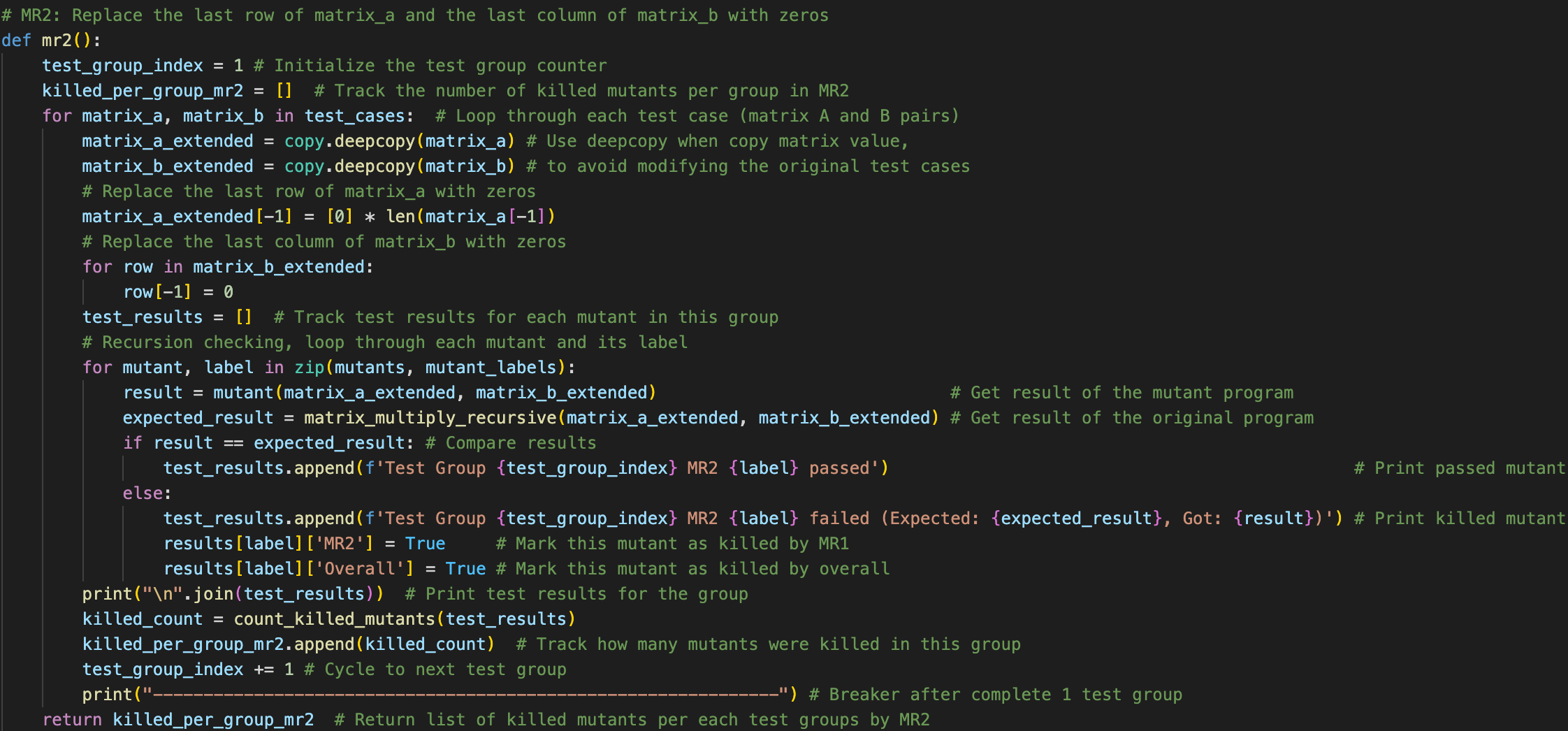
**MR1 Implementation:**

We implemented MR1 by scaling all matrix elements by a factor of 2. We compared the output of the mutant implementations with the expected result generated by the original algorithm applied to the scaled matrices.

**Program:**

**MR2 Implementation:**

For MR2, we replaced the last row of matrix A and the last column of matrix B with zeros and tested each mutant for correctness against the expected zeroed result.

**Program:**

**4. Mutant Generation**

Mutation testing involves introducing small changes (mutants) into the original program and checking if the metamorphic relations detect these faults. A mutant is considered "killed" if it violates any of the metamorphic relations.

Mutation operators that has been conducted in this program assessment includes Arithmetic Operators Replacement, Relational Operator Replacement, Logical Operator Replacement, Constant Replacement, Variable Replacement, Change of array index, Change of control flow, and Change a variable to a constant.

This table illustrates the implementation of 30 mutant programs, comparing their original and mutant algorithms, alongside with the mutation program ID and description.

**5. Test Groups**

We conducted **5 test groups** (set of test cases grouping matrix A and B) of matrices for both MR1 and MR2. Each test group involved running the original and mutant programs through the metamorphic relations and observing if any violations (differences) occurred.

test\_cases = [

([[1, 2], [3, 4]], [[5, 6], [7, 8]]), # Test group 1

([[9, 8], [7, 6]], [[5, 4], [3, 2]]), # Test group 2

([[0, 0], [1, 1]], [[3, 3], [4, 4]]), # Test group 3

([[0, 9], [1, 8]], [[2, 7], [3, 6]]), # Test group 4

([[1, 1], [1, 1]], [[7, 7], [7, 7]]) # Test group 5

]

### **6. Results**

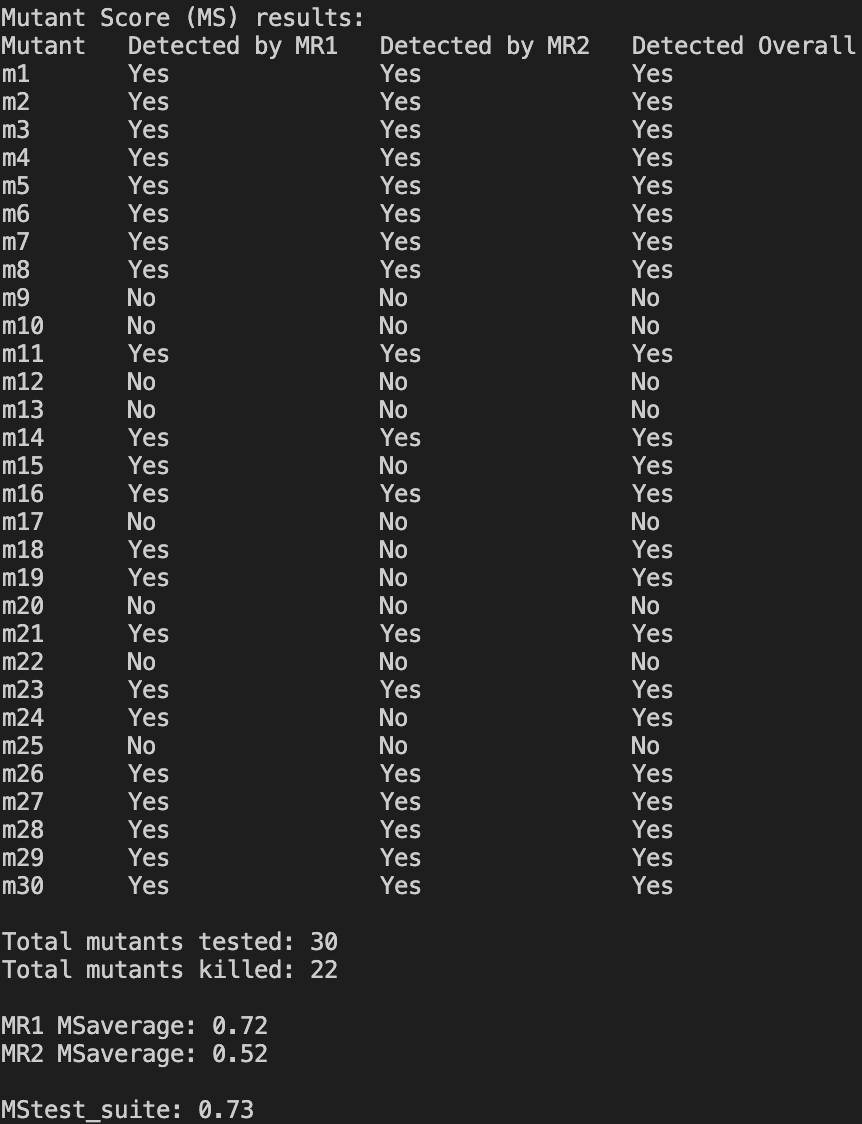
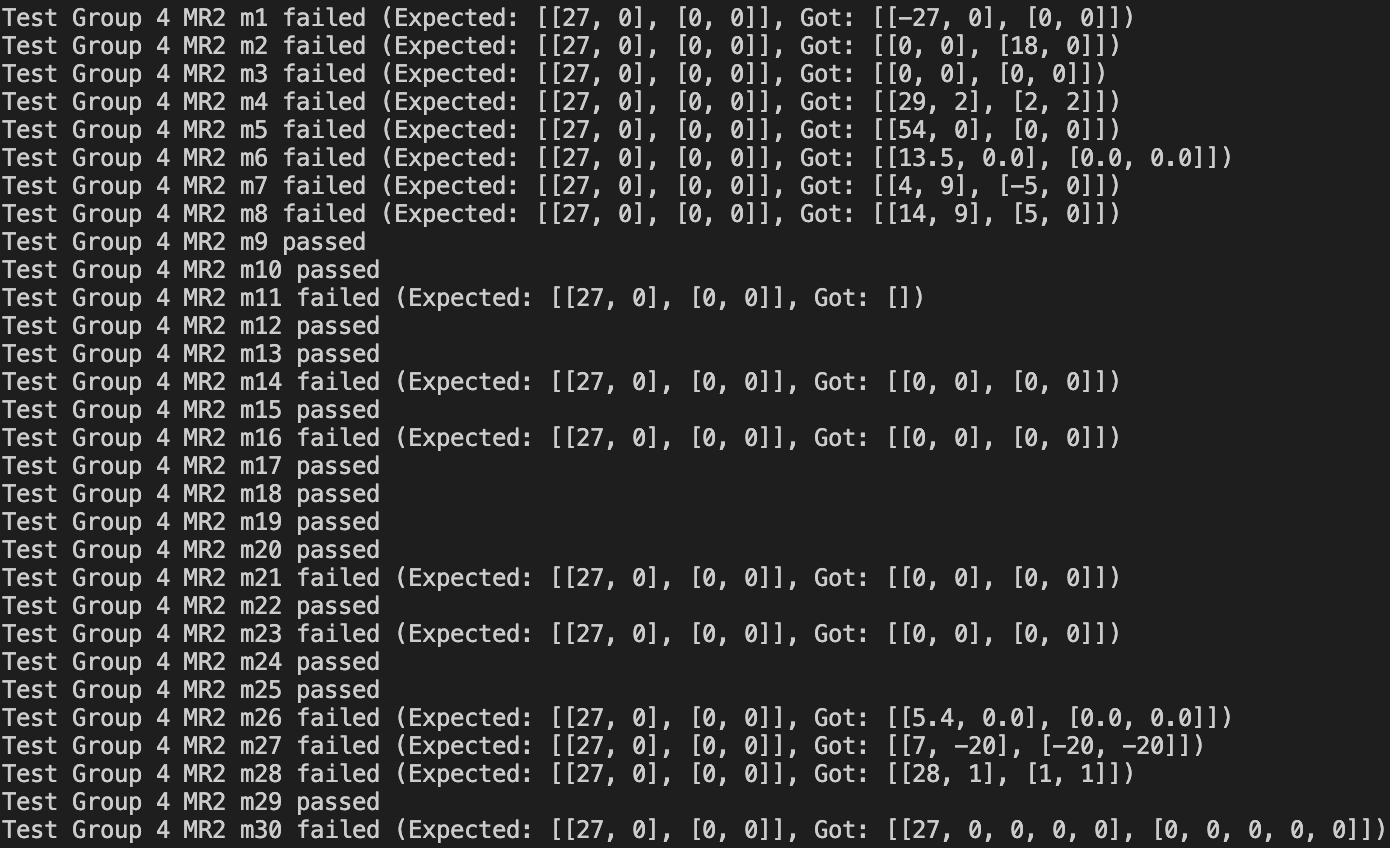
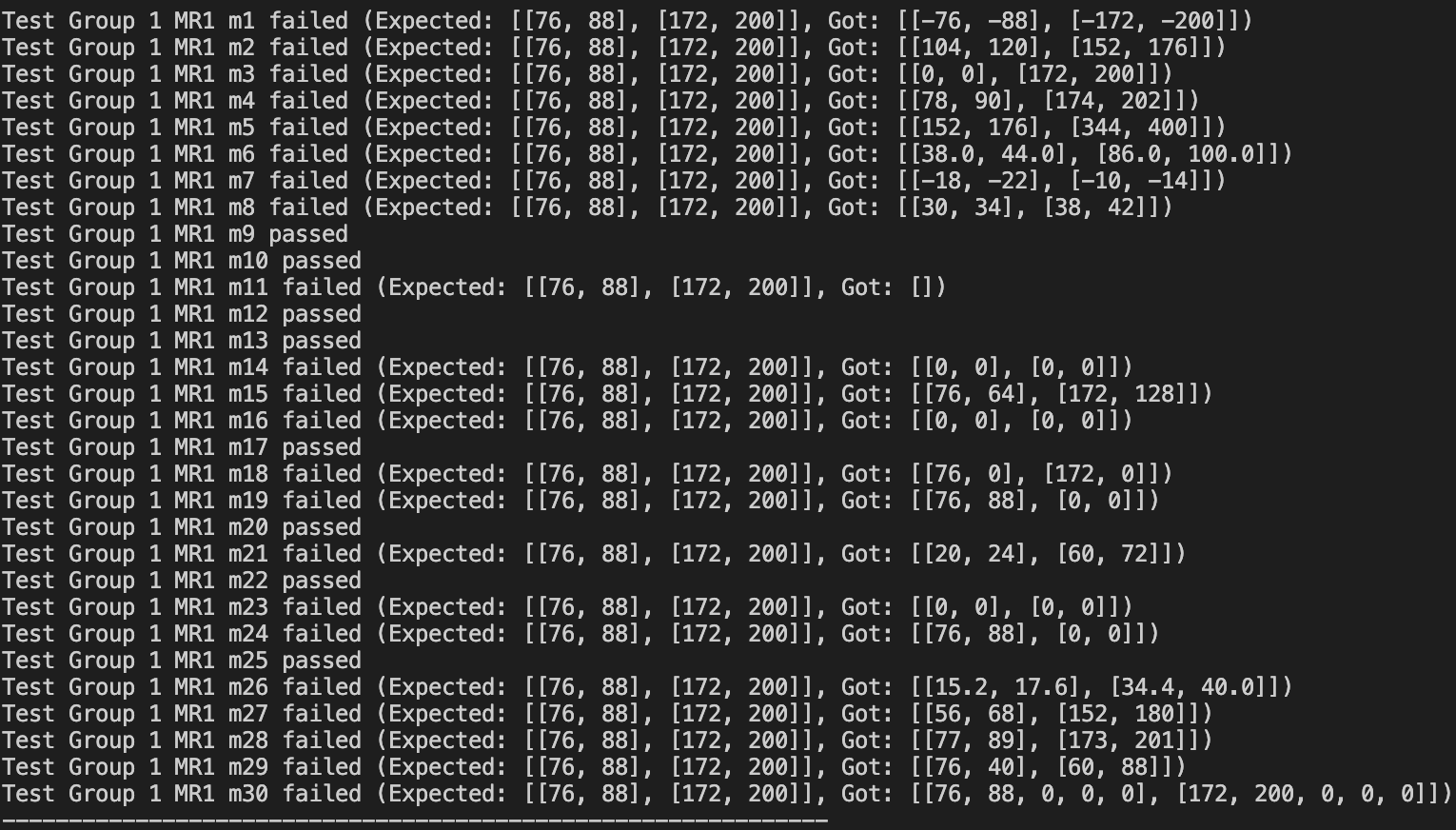
Screenshots demonstrating program result execution, including mutation score (MS) results for table list of 30 mutants, indicating whether each mutant was detected by MR1, MR2, or overall; and output of the metamorphic testing process for following test groups with MR1 and MR2, showing detailed results for each mutant, specifying whether the mutant passed or failed, along with the expected and actual outputs for failed mutants.

Figure. Test result output showing mutation detection results for MR1 in Test Group 1.

Figure. Test result output showing mutation detection results for MR2 in Test Group 4.

Figure. Mutation score results showing MR1 and MR2 detection effectiveness across 30 mutants.

**Evaluation Metrics:**

Formula applied to calculate average mutation score (MSaverage) in this program:

* MSaverage = =
* With “total\_killed\_mri” to be total number of mutant killed by each MR.

Formula applied to calculate overall mutation score generally for the program (MStest\_suite):

* MStest\_suite = =
* With “total\_mutants\_killed” to be total number of mutant killed either from MR1 and/or MR2.

Terminologies used:

* MR1 MSaverage: The proportion of mutants killed by MR1 over the total mutants tested.
* MR2 MSaverage: The proportion of mutants killed by MR2 over the total mutants tested.
* MStest\_suite: The overall mutation score, representing the proportion of mutants killed by either MR1 or MR2.

#### **Mutation Score Summary:**

The table below summarizes the detection rates for each mutant.

* **Total mutants tested: 30**
* **Total mutants killed: 22**
* **MR1 MSaverage**: 0.72
* **MR2 MSaverage**: 0.52
* **MStest\_suite**: 0.73

**7. Discussion**

**Effectiveness of Metamorphic Relations:**

* **MR1 (Scaling)**: This metamorphic relation proved highly effective in detecting arithmetic errors. It caught 72% of mutants, showcasing its strength in validating the scalar multiplication property of the algorithm.
* **MR2 (Zeroing)**: MR2 was less effective overall, detecting 52% of mutants. However, it uniquely detected some mutants missed by MR1, primarily catching mutants where the zero replacement did not propagate correctly.

**Complementarity of MRs:**

The combination of MR1 and MR2 achieved a high overall mutation score of 73%. Both MRs demonstrated some overlap, but they also caught unique mutants. MR1 detected many mutations related to arithmetic operator replacements, while MR2 was more sensitive to constant changes and control flow adjustments.

### **8. Conclusion**

Metamorphic testing using the proposed relations MR1 (Scaling the Matrix) and MR2 (Zeroing Last Row and Column) was successful in identifying a wide range of logical and arithmetic errors in the matrix multiplication algorithm. The combined mutation score of 73% demonstrates the effectiveness of this approach.