

SWE30009- Software Testing and Reliability

Project Report

Name: Pham Do Tien Phong

Student ID: 104189767

Task 1: Random testing

Subtask 1.1:

Intuition of random testing:

Random testing (RT) is a black-box technique that randomly selects test cases from the entire input domain, based on the assumption that random inputs are more likely to detect failures.

This makes exhaustive testing of complex systems feasible. RT can be done:

- With replacement: previously selected inputs could be selected again
- Without replacement: previously selected inputs could not be selected again

While traditional random testing relies on static input distributions, Adaptive Random Testing (ART) improves upon RT by dynamically adjusting test inputs based on previous results. Studies show that ART requires 50-60% fewer test cases to detect the first failure compared to RT with replacement.

Distribution profiles for random testing:

The effectiveness of random testing hinges on the distribution of test inputs. There are 2 traditional approaches to random testing:

- Uniform distribution: each input is of equal selection probability
- Operational distribution (profiles): selection probability follows the probability of being used

ART takes this concept further by dynamically adjusting the distribution based on test outputs, increasing its effectiveness and probability of detecting failure.

1. Fixed-Size-Candidate-Set ART (FSCS-ART): Generate random candidates, pair each with its closest executed test case, and select the candidate furthest from its nearest test case using the maximin criterion. This maximizes test coverage by exploring untested areas.
2. ART by exclusion: Establish exclusion regions for each executed test case. Generate random candidates until one falls outside all exclusion regions, ensuring new test cases target untested areas for enhanced coverage and diversity.
3. ART by random partition: Divide the input domain into partitions using the most recent test case. Select the largest partition and choose a random input from it as the

next test case. This maximizes coverage by selecting diverse test cases from the input domain.

Process of random testing and some applications:

The process of random testing has 4 simple steps:

1. Define input domain: Determine the size of possible input values for the system
2. Generate random inputs: Create test inputs randomly within input domain size
3. Execute test case: Run the system with the generated inputs
4. Analyze outputs: Compare the actual output with expected output to detect failure
5. Iterative refinement: Repeat the process with different random inputs

Random testing can be applied at various stages of software development:

- Unit testing: Testing individual software components
- Integration testing: Testing interaction between components
- System testing: Testing entire system
- Performance testing: Identifying performance bottlenecks
- Security testing: Discovering vulnerabilities

Examples/ Illustrations:

A simple calculator application can be tested using random numbers for operands and operators to verify the correctness of calculations. By generating random number inputs for '+', '-', '*', '/', different combinations of inputs can be failure-causing inputs.

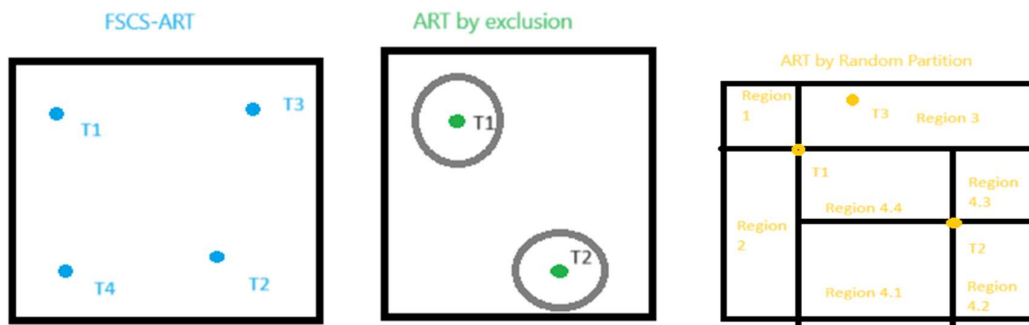


Figure 1: Illustration of ART

Subtask 1.2:

Test Cases	Input	Expected output
Small positive duplicate numbers	15, 2, 44, 1, 33, 1, 44, 18 3,9	1, 1, 2, 3, 9, 15, 18, 33, 44, 44
Small negative duplicate numbers	-99, -102, -100, -2, -3, -4, -40, -90, -100, -99, -4	-102, -100, -100, -99, -99, -90, -40, -4, -4, -3, -2

All positive duplicate numbers	1,1,1,1,1,1,1,1,1,1,1,1,1	1,1,1,1,1,1,1,1,1,1,1,1,1
All negative duplicate numbers	-100,-100,-100,-100,-100,-100,-100	-100,-100,-100,-100,-100,-100,-100
Large positive duplicate numbers	100000,222222,90923092,1231233,23323,57475784,3333,3333,222222	3333,3333,23323,100000,222222,222222,1231233,57475784,90923092
Large negative duplicate numbers	-23232323,-1102030,-12010230,-0202020,-1102030,-555552,-555552,-10202	-23232323,-12010230,-1102030,-1102030,-555552,-555552,-202020,-10202
All possible values	123123213,239239923,1,23320,0,-23344,0,-1111,34,-9999999,23,1,-1111	-9999999,-23344,-1111,-1111,0,0,1,1,23,34,23320,123123213,239239923

Methodology

Random list generation: A random list is used to create a list of integers. The list length and the range of integer values can be varied for different test cases

Duplicated integers: To test duplicated integers, I randomly select integers from generated list and add them multiple times

Expected output analysis: The expected output is simply sorted input integers

By generating various random test cases, we can increase the likelihood of detecting potential failures.

Task 2: Metamorphic testing

Subtask 2.1:

Test oracle & untestable systems:

A test oracle is a mechanism or procedure against which the computed outputs could be verified, such as an inverse function or a backward substitution. Testers can consider some test oracle problems, including the absence of test oracles or available test oracles that are too expensive to be applied.

An untestable program refers to a program that is not untestable or non-testable if the outputs of some inputs cannot be verified (or cannot be verified in practice). For example, a weather forecasting system which reports the amount of rain, a clinical x-ray system, or an earthquake warning system.

The motivation and intuition of metamorphic testing and metamorphic relations

Metamorphic testing (MT) addresses challenges in testing untestable systems by creating test cases based on properties and reducing test oracle problems. While traditional testing demonstrates the presence of faults, MT focuses on relationships between inputs and outputs,

rather than precise expected outputs. In MT, we understand connections between inputs and outputs without knowing the exact accuracy for specific inputs.

Metamorphic relations (MRs) are key algorithm characteristics, linking interconnected inputs and outputs. MRs encompass more than simple identity and numeric relations, covering diverse algorithm properties. MT involves executing multiple tests where follow-up cases depend on previous outputs, ensuring robustness across various scenarios

Metamorphic groups of inputs, or simply metamorphic groups (MGs), are sets of inputs that consist of the source test cases and their associated follow-up test cases in a given MR. An MR can have many metamorphic groups, which may be satisfied with some MGs but can be violated with some other MGs

The outcome of applying MT is either

- Satisfaction of MR with respect to a MG: not ensuring the accuracy of the program;
- or
- Violation of MR with respect to a MG: ensuring the program is faulty

The process of metamorphic testing and some applications:

Metamorphic testing process (A simplified form):

1. Define and execute test cases by utilizing several test case selection strategies
2. Identify some properties of the problem (based on metamorphic relations)
3. Build and implement follow-up test cases from the source test cases, depending on the identified metamorphic relations
4. Verify the MRs using the computed outputs

Some applications:

- GraphicsFuzz uses MT as its main technique to test graphics drivers
- Testing Web enabled simulation at Scale using Metamorphic Testing
- Metamorphic Model-Based Testing Applied on NASA DAT – an Experience Report

Example/Illustration

- \cos function has the corresponding properties
- $\cos(-x) = \cos(x)$
- Suppose $x = 49.1$
- Compute -49.1
- Execute the program with -49.1 as input
- Check $\cos(-49.1) = \cos(49.1)$

Figure 2: Example of Metamorphic testing

- To find the sum of a series of integers
- Suppose the selected input is: SI = [10, 15, 11, 2, 100]
- Execute duplication input : FI = [10, 10, 15, 15, 11, 11, 2, 2, 100, 100]
- Check the FO = $2 * SO = 2 * 138 = 276$

Figure 3: Example of Metamorphic testing by the duplication method

Find the short test path SP(G,a,b) which returns a path from node a to node b in graph G

- Consider the property: $|SP(G,a,b)| = |SP(G,b,a)|$
- Execute SP with input (G,b,a)
- Check whether $|SP(G,a,b)| == |SP(G,b,a)|$

Figure 4: Example of Metamorphic testing

Subtask 2.2:

Metamorphic relation 1: Order preservation

Description: The relative order of distinct elements should be preserved after sorting. If an element A appears before element B in the unsorted list, and both A and B are distinct, then A should still appear before B in the sorted list.

Metamorphic group: To evaluate this relation, I create a set of test cases by applying various modifications to the source input list :

- Source input: [-32, 7, 0, 45, -12, 7, 99, 0]
- Following test case:
 - Swap randomly selected elements: [-32, 45, 0, 7, -12, 7, 99, 0]
 - Reverse a sublist: [-32, 7, 0, 99, 45, -12, 7, 0]
 - Add a random elements: [-32, 7, 0, 23, 45, -12, 7, 99, 0]
 - Delete a random element: [-32, 7, 0, 45, -12, 7, 99]

Explanation: By comparing the relative positions of distinct elements in the sorted output of these test cases, testers can ensure if the order preservation property holds.

Metamorphic relation 2: Idempotence

Description: Applying the sorting algorithm twice to the same input list should produce the same output. This property indicates that the sorting algorithm has reached a stable state.

Metamorphic group:

- Source input: [3, 7, 2, 9, 5]
- Following test cases
 - Add duplicated elements: [3, 7, 2, 9, 5, 3, 7, 2]
 - Reversing list: [9, 5, 2, 7, 3]

Explanation: By applying the sorting algorithm to the source, following test cases, and comparing the outputs, we can verify if the idempotence property holds.

Conclusion

These two MRs provide a foundation for testing the integer sorting program. By generating diverse test cases based on these relations, testers may enhance the effectiveness of detecting potential failures in the sorting algorithm.

Subtask 2.3:

	Random testing	Metamorphic testing
Oracle	Typically, this requires a correct oracle to determine if a test case passes or fails. Without a defined oracle, determining the correctness of outputs can be challenging, potentially leading to difficulties in result evaluation and test case validation.	This depends on metamorphic relations instead of a correct test oracle. This is a simple but effective method to alleviate the test oracle problem. Identifying appropriate metamorphic relations and establishing reliable oracles may require domain expertise and additional effort, potentially complicating the testing process.

Test case	Test cases are generated randomly within the input domain. This can lead to a high number of test cases to achieve efficiency. The random selection of test case generation may lead to inefficient coverage of critical system areas and can make reproducing specific scenarios difficult, affecting the effectiveness of results.	Test cases are executed from source test cases using metamorphic relations. This can result in a smaller number of test cases while still achieving good coverage. Defining appropriate metamorphic relations can be complex, requiring a thorough understanding of system behavior, and the process may introduce additional overhead in test case generation
Advantage	1. Intuitively simple 2. Allow statistical, quantitative estimation of the software's reliability _Easy to detect expected failure	1. Effective for untestable systems 2. Can achieve high fault detection efficiency. 3. Can be combined with other testing techniques.
Disadvantage	1. Ineffective (not using any information to guide the selection strategies of the cases) 2. Resources are intensive, especially when a large number of test cases need to be generated and executed 3. It requires a precise oracle for each test case.	1. Requires identification of suitable metamorphic relations. 2. Implementing metamorphic testing can be more complex than random testing. 3. It may not be applicable to all types of systems, particularly those where clear input-output relationships are harder to define.

In summary, Random testing is straightforward but can be inefficient and resource-intensive, while Metamorphic testing is efficient for certain scenarios but requires more initial setup and complexity.

Task 3: Test a program of your choice

```

1  # Original Merge Sort Implementation
2  def merge_sort(arr):
3      if len(arr) > 1:
4          mid = len(arr) // 2
5          left_half = arr[:mid]
6          right_half = arr[mid:]
7          # Recursive call on each half
8          merge_sort(left_half)
9          merge_sort(right_half)
10         i = j = k = 0
11         # Copy data to temp arrays left_half and right_half
12         while i < len(left_half) and j < len(right_half):
13             if left_half[i] < right_half[j]:
14                 arr[k] = left_half[i]
15                 i += 1
16             else:
17                 arr[k] = right_half[j]
18                 j += 1
19             k += 1
20         # Checking if any element was left
21         while i < len(left_half):
22             arr[k] = left_half[i]
23             i += 1
24             k += 1
25         while j < len(right_half):
26             arr[k] = right_half[j]
27             j += 1
28             k += 1
29         return arr

```

Figure 5: Screenshot of program

The real-world program selection:

I focus on the real-world factor of the program, which means the selected program is not only easy to apply in the real scenarios but also suitable for demonstration of metamorphic testing. The provided code implements the Merge Sort algorithm, a highly efficient and widely used sorting technique based on the divide-and-conquer paradigm. The function `merge_sort` recursively divides the input array into two halves until each subarray contains a single element or is empty. It then merges these subarrays back together in a sorted order. This merging process involves

comparing elements from two subarrays and copying the smaller element into the original array, ensuring that the final array is sorted.

Real-world scenarios for the program:

Database Management sorting records: Efficiently sort large datasets in databases to facilitate quick searching, filtering, and indexing.

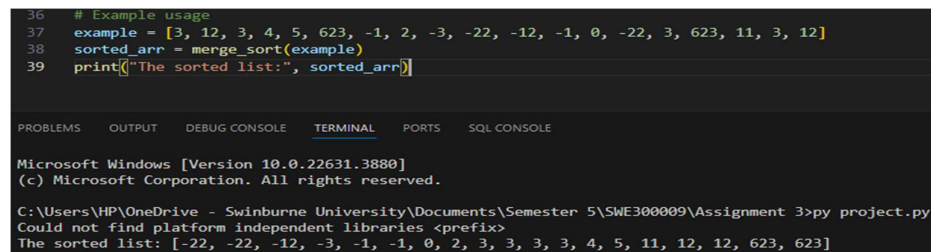
Data Analysis processing of large data sets: Sort data for analysis, such as survey results, log data, or any large collection of records.

Financial Systems transaction processing: Sort transactions, trades, or other financial records for reporting, analysis, or auditing purposes.

Requirement satisfaction:

1. The original program under test is implemented correctly and obtained from GitHub via this [Link](#)
2. Programming language: The program is written in Python
3. It is neither too difficult and complex nor too simple

Screenshot of program executing:



```

36 # Example usage
37 example = [3, 12, 3, 4, 5, 623, -1, 2, -3, -22, -12, -1, 0, -22, 3, 623, 11, 3, 12]
38 sorted_arr = merge_sort(example)
39 print("The sorted list:", sorted_arr)

```

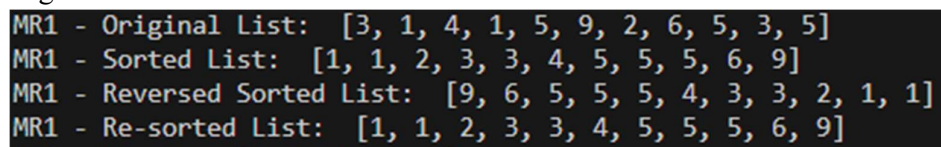
Microsoft Windows [Version 10.0.22631.3880]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive - Swinburne University\Documents\Semester 5\SWE300009\Assignment 3>py project.py
Could not find platform independent libraries <prefix>
The sorted list: [-22, -22, -12, -12, -3, -1, -1, 0, 2, 3, 3, 3, 3, 4, 5, 11, 12, 12, 623, 623]

Figure 6: Screenshot of program executing

Metamorphic Relations:

- MR1: Reversing and Sorting tests the idempotence of sorting algorithms. If you sort a list and then reverse the sorted list, sorting the reversed list should give you back the original list.



```

MR1 - Original List: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
MR1 - Sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
MR1 - Reversed Sorted List: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
MR1 - Re-sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

```

Figure 7: Executing MR 1

- MR2: Concatenating and Sorting checks commutativity of the sorting function. Sorting a concatenation of two lists should yield the same result as concatenating the individually sorted lists.

```
MR2 - List 1: [1, 2, 3, 4, 5]
MR2 - List 2: [6, 7, 8, 9, 10]
MR2 - Concatenated and Sorted List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
MR2 - Individually Sorted and Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure 8: Executing MR2

- MR3: Doubling the list checks the stability and correctness of the sorting function when handling duplicates. If you double the list by concatenating it with itself, the sorted result should contain each element from the original sorted list twice, in the same order.

```
MR3 - Original List: [2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]
MR3 - Sorted List: [-111111, -22, 5, 13, 22, 111, 129, 335, 2323, 11126, 22335]
MR3 - Doubled List: [2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5, 2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]
MR3 - Sorted Doubled List: [-111111, -111111, -22, -22, 5, 5, 13, 13, 22, 22, 111, 111, 129, 129, 335, 335, 2323, 2323, 11126, 11126, 22335, 22335]
MR3 - Expected Sorted Doubled List: [-111111, -111111, -22, -22, 5, 5, 13, 13, 22, 22, 111, 111, 129, 129, 335, 335, 2323, 2323, 11126, 11126, 22335, 22335]
```

Figure 9: Executing MR3

- MR4: Adding a constant checks the translation invariance of the sorting function. If you add a constant to every element of the list and sort it, the sorted list should maintain the same relative order but with each element increased by the constant.

```
MR4 - Original List: [183, 11122, 234412, -1123, -50022, -90238, -223, -16, 2135, 323, -12391203]
MR4 - Sorted List: [-12391203, -90238, -50022, -1123, -223, -16, 183, 323, 2135, 11122, 234412]
MR4 - Modified List: [193, 11132, 234422, -1113, -50012, -90228, -213, -6, 2145, 333, -12391193]
MR4 - Sorted Modified List: [-12391193, -90228, -50012, -1113, -213, -6, 193, 333, 2145, 11132, 234422]
MR4 - Expected Sorted Modified List: [-12391193, -90228, -50012, -1113, -213, -6, 193, 333, 2145, 11132, 234422]
All metamorphic tests passed for the original function.
```

Figure 10: Executing MR4

Summary Table for MRs

MR	Original List	Transformed List	Sorted List	Result
MR1	[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]	[9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]	[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]	PASS
MR2	[1, 2, 3, 4, 5] + [6, 7, 8, 9, 10]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	PASS
MR3	[2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]	[2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5, 2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]	[-111111, -111111, -22, -22, 5, 5, 13, 13, 22, 22, 111, 111, 129, 129, 335, 335, 2323, 2323, 11126, 11126, 22335, 22335]	PASS
MR4	[183, 11122, 234412, -1123, -50022, -90238, -223, -16, 2135, 323, -12391203]	[193, 11132, 234422, -1113, -50012, -90228, -213, -6, 2145, 333, -12391193]	[-12391193, -90228, -50012, -1113, -213, -6, 193, 333, 2145, 11132, 234422]	PASS

- Original List:** The initial list provided for each metamorphic relation test.
- Transformed List:** The list after applying the specific transformation (e.g., reversing, adding a constant, etc.).
- Sorted List:** The list after sorting the transformed list.
- Result:** Whether the sorted transformed list matches the expected result, indicating if the metamorphic relation test passed or failed.


```

Testing original merge sort:
MR1 - Original List: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
MR1 - Sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
MR1 - Reversed Sorted List: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
MR1 - Re-sorted List: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
MR2 - List 1: [1, 2, 3, 4, 5]
MR2 - List 2: [6, 7, 8, 9, 10]
MR2 - Concatenated and Sorted List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
MR2 - Individually Sorted and Concatenated List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
MR3 - Original List: [2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]
MR3 - Sorted List: [-111111, -22, 5, 13, 22, 111, 129, 335, 2323, 11126, 22335]
MR3 - Doubled List: [2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5, 2323, 111, -22, -111111, 22335, 129, 22, 11126, 335, 13, 5]
MR3 - Sorted Doubled List: [-111111, -111111, -22, -22, 5, 5, 13, 13, 22, 22, 111, 111, 129, 129, 335, 335, 2323, 2323, 11126, 11126, 22335, 22335]
MR3 - Expected Sorted Doubled List: [-111111, -111111, -22, -22, 5, 5, 13, 13, 22, 22, 111, 111, 129, 129, 335, 335, 2323, 2323, 11126, 11126, 22335, 22335]
MR4 - Original List: [183, 11122, 234412, -1123, -50022, -90238, -223, -16, 2135, 323, -12391203]
MR4 - Sorted List: [-12391203, -90238, -50022, -1123, -223, -16, 183, 323, 2135, 11122, 234412]
MR4 - Modified List: [103, 11132, 234422, -1113, -50012, -90228, -213, -6, 2145, 333, -12391193]
MR4 - Sorted Modified List: [-12391193, -90228, -50012, -1113, -213, -6, 193, 333, 2145, 11132, 234422]
MR4 - Expected Sorted Modified List: [-12391193, -90228, -50012, -1113, -213, -6, 193, 333, 2145, 11132, 234422]
All metamorphic tests passed for the original function.

```

Figure 11: Executing of all MRs

Mutants Generation and Evaluation:

Mutants No	Mutants Description	MR1	MR2	MR3	MR4
1	Replace '<' with '<=' in comparison	Survived	Survived	Survived	Survived
2	Change '// 2' to '// 3' in middle index calculation	Killed: Incorrect partitioning	Killed: Incorrect partitioning	Killed: Incorrect partitioning	Killed: Incorrect partitioning
3	Replace '>' with '>=' in length check	Killed: Infinite recursion due to incorrect length check	Killed: Infinite recursion due to incorrect length check	Killed: Infinite recursion due to incorrect length check	Killed: Infinite recursion due to incorrect length check
4	Replace '<' with '>' in comparison	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering
5	Swap 'left_half' and 'right_half' only in the recursive call order	Survived	Survived	Survived	Survived
6	Do not merge the left half	Killed: Only right half considered	Killed: Only right half considered	Killed: Only right half considered	Killed: Only right half considered
7	Replace 'k += 1' with 'k -= 1'	Killed: Infinite loop	Killed: Infinite loop	Killed: Infinite loop	Killed: Infinite loop
8	Always return the original array	Killed: No sorting performed	Killed: No sorting performed	Killed: No sorting performed	Killed: No sorting performed
9	Replace 'if left_half[i] < right_half[j]' with 'if left_half[i] <= right_half[j]'	Survived	Survived	Survived	Survived
10	Change 'mid = len(arr) // 2' to 'mid = (len(arr) // 2) + 1'	Survived	Survived	Killed: Incorrect partitioning	Survived

11	Always merge right_half elements	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering
12	Always sort the entire array instead of splitting	Survived	Survived	Survived	Survived
13	Do not merge the left half	Killed: Only right half considered	Killed: Only right half considered	Survived	Killed: Only right half considered
14	Incorrect mid calculation using addition	Killed: Incorrect partitioning	Killed: Incorrect partitioning	Survived	Survived
15	Always select right_half element in merge	Killed: Only right half considered	Killed: Only right half considered	Survived	Survived
16	Always select left_half element in merge	Killed: Only left half considered	Killed: Only left half considered	Killed: Incorrect ordering	Survived
17	Incorrect index increment in merge	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Survived
18	Double the index increment in merge	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Survived
19	Always select left_half element 'if i < len(left_half)'	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Survived
20	Always select right_half element 'if j < len(right_half)'	Killed: Incorrect ordering	Killed: Incorrect ordering	Killed: Incorrect ordering	Survived
	Score	75%	75%	65%	45%

Discussion and Conclusion

The analysis of the mutants indicates that MR1 and MR2 are the most effective metamorphic relations for detecting faults in the merge sort algorithm. They successfully identified 75% of the introduced mutants. MR3 also showed a relatively high detection rate at 65%, while MR4 was the least effective with a 45% detection rate.

Overall, these findings suggest that incorporating multiple metamorphic relations is beneficial for robust fault detection in sorting algorithms. However, reliance solely on MR4 would be insufficient due to its lower effectiveness. Therefore, it is recommended to use a combination of MR1, MR2, and MR3 to ensure comprehensive fault detection and validation of the merge sort algorithm's correctness.

*GitHub link: [Program Link](#)