

Final Project Report

Task 1: Random Testing

Subtask 1.1: Understanding Random Testing Methodology

Intuition of Random Testing

Random testing is a black-box software testing technique where test cases are generated using random independent inputs. The idea is to explore the input space of a program by randomly selecting inputs to identify defects that may not be found through systematic or manual testing methods. This approach is based on the principle that random inputs can uncover unanticipated edge cases and bugs (Chen et. al., 2007).

Distribution Profiles for Random Testing

Type	Description	Example
Uniform Distribution	Every possible input has an equal chance of being selected.	In a calculator program using a uniform distribution, we can ensure that every number within the specified range is equally likely to be selected for testing. This approach helps to uncover issues across the entire range
Normal Distribution (a type of non-uniform distribution)	Inputs are chosen based on a bell curve, where values near the mean are more likely selected	In a loan calculator application typically calculates monthly payments, total interest, and other financial metrics, the real-world loan amounts and interest rates often cluster around certain typical values, making normal distribution an ideal choice for testing, emphasizing on the input range with higher probability
Custom Distribution (a type of non-uniform distribution)	Tailored distributions to match specific input patterns or real-world usage	When testing an e-commerce website's discount code validation, based on real-world usage patterns, certain discount codes are used more frequently, inputs might include commonly used codes (e.g., "SAVE10", "FREESHIP") more often than infrequent codes.

Process of Random Testing

Based on Priya (2023), the random testing process may comprise of:

1. Identifying the range and types of inputs (input domain).
2. Using random number generators to produce inputs within the defined space.
3. Running the program with the generated test cases.
4. Verifying if the outputs are correct using an oracle or predefined criteria.
5. Recording the outcomes to analyze and make modifications.

Applications of Random Testing

- **Testi website capability to work under stress:** Utilizing random testing simplicity characteristics. It can simulate high load scenarios with massive amount of test inputs, providing insights into how systems perform under stress and identifying bottlenecks.

- **Uncover edge cases in a form input of log in form:** Random testing introduces inputs that are not anticipated by developers, making it highly effective at revealing defects that systematic testing might miss.
- **Network Protocol Testing:** Send randomly generated and malformed messages to the system using a custom network protocol to test robustness against unexpected inputs. This verifies the protocol's ability to handle edge cases and security vulnerabilities.

Subtask 1.2: Generating Test Cases for a Sorting Program

Program Description:

A program designed to sort a non-empty list of integer numbers, which may contain duplicates.

Random Test Cases:

To automate the generation of random test cases for a sorting program, we can create a Python script that randomly generates lists of integers within a specified range and length. First, we define the range of integers and the range of list lengths, then use a random number generator to create lists.

To improve the efficiency and effectiveness of random testing, we can partition the input domain into different categories and generate random test cases within each partition. This ensures that we test a more representative sample of inputs, including edge cases and typical usage scenarios. For this specific case, assume the program accepts integers from range between -2^{32} to 2^{32} (any wider or narrower range works since we exclusively concentrate on the testing method).

The array input length may also be randomized, therefore in this report, the test cases end with semicolon, meaning potentially there might be more elements.

Domain partition	Test case
Small positive numbers	[3, 45, 67, 21, 89, ...]
Small negative numbers	[-64, -38, -99, -12, -55, ...]
Large positive numbers	[2147483649, 4294967294, 250034930, 3038495060, ...]
Large negative numbers	[-4294967294, -2500012600, -3001294728, -2147483649, ...]
All positive numbers	[2343298, 153842390, 98, 32394239, 73203947, 5342038, ...]
All negative numbers	[-4294967294, -212, -25, -2147483649, -2324398, ...]
All possible value	[21209382, -72930219, 382, 2190381, -48, 29 -121238243, ...]

Task 2: Metamorphic Testing

Subtask 2.1: Understanding Metamorphic Testing Methodology

Test Oracle

In traditional software testing, a test oracle typically provides a clear expected result to compare against the actual result of a test case.

In metamorphic testing, since metamorphic testing can be used to test module with non-deterministic behavior, a test oracle is interpreted as the mechanism through which the correctness of the software is indirectly verified by examining the relationships between inputs and outputs rather than by directly validating the output against a known correct result.

Untestable systems

An untestable system is one for which it is challenging or impossible to determine whether the system behaves correctly based on the given inputs and expected outputs, since:

Reason for un-testability	Example
There's no clear way to verify the correctness of the outputs	Machine learning and optimization models
The conditions are complex or context-dependent	Financial trading system, whose environment involves market conditions, and unpredictable variables
Outputs vary even under the same conditions	Weather simulation model, where the same simulation produces different results since all answers have a chance of being true
Relationships between inputs, outputs are hard to establish	Music-generate AI, the relationship between input (genre, tempo) and generated music is unclear

Motivation

Metamorphic testing is based on the concept that understanding the relationship between different outputs of a program is often simpler than comprehending its input-output behavior in isolation. (Hai, 2024). Therefore, the motivation for metamorphic testing lies in its ability to address the challenges of testing complex, non-deterministic, and poorly specified systems by focusing on the properties and relationships inherent in the system.

Intuition

The intuition behind metamorphic testing is to leverage inherent properties and relationships within a system to validate its behavior, especially when traditional methods fall short. It describes how changes to inputs should logically affect outputs, this approach focuses on relative correctness rather than exact outputs. It generates follow-up test cases from initial ones, ensuring various related inputs are tested to uncover inconsistencies. This method is effective for complex, nondeterministic systems where precise test oracles are unavailable.

Metamorphic Relations

A metamorphic relation (MR) is a rule describing how the output of a system should change in response to specific changes in the input. In metamorphic testing, MRs are used to generate follow-up test cases from an initial test case by modifying the input based on these relations. This method ensures consistent behavior across different but related inputs, bypassing the need for precise expected outputs and improving test coverage. For example, in a sorting algorithm, an MR would state that reversing a list and sorting it again should yield the same sorted list. MRs are crucial for testing complex and nondeterministic systems where defining exact expected results is challenging (Hai, 2024).

Process of Metamorphic Testing:

1. Determine relevant MRs for the program.
2. Select Source Test Cases.
3. Generate Follow-up Test Cases.
4. Run the program with both source and follow-up test cases.
5. Compare Outputs.
6. Identify deviations between the expected and actual outputs of follow-up test cases.
7. Modify the program if needed to start another test process.

Applications of Metamorphic Testing:

- **Testing Machine Learning Models:** Ensuring consistent performance across variations.
- **Validating Weather Prediction Software:** Checking that changes in input parameters produce expected variations in results.
- **Sort-list program:** one MR could be sorting a list twice should yield the same result.

Subtask 2.2: Applying Metamorphic Testing to a Sorting Program

Metamorphic Relation 1: Sorting Idempotence

Description: Sorting a list that has already been sorted should yield the same list.

Intuition: If a list is correctly sorted, sorting it again should not change it.

Concrete Metamorphic Group:

- Source test case input: [1, 4, 2, 5, 3]
- Source test case output: [1, 2, 3, 4, 5]
- Follow-up test case: [1, 2, 3, 4, 5] (result of sorted [1, 4, 2, 5, 3])
- Check that sorting [1, 2, 3, 4, 5] again results in [1, 2, 3, 4, 5].

Metamorphic Relation 2: Permutation Invariance

Description: Sorting a list should produce the same result regardless of the initial order.

Intuition: The result should depend only on the set of elements, not their initial order.

Concrete Metamorphic Group:

- Source test case input: [3, 1, 4, 2, 5]
- Source test case output: [1, 2, 3, 4, 5]
- Follow-up test case: [1, 4, 3, 2, 5] (permutation of source test case)
- Check that sorting [1, 4, 3, 2, 5] again results in [1, 2, 3, 4, 5].

Metamorphic Relation 3: Adding a Duplicate Element

Description: Adding a duplicate of an existing element to a sorted list should maintain the correct order when sorted again.

Intuition: The algorithm should handle duplicates without disrupting the sorted order.

Concrete Metamorphic Group:

- Source Test Case: [4, 2, 3, 2, 5]
- Source test result: [2, 2, 3, 4, 5]
- Follow-up Test Case: [4, 2, 3, 2, 4, 5, 2] (source test case with duplication of elements)
- Check that sorting [4, 2, 3, 2, 4, 5, 2] results in [2, 2, 2, 3, 4, 4, 5] the order is like source case whereas there exist more duplicated elements.

Metamorphic Relation 4: Adding an Element at the End

Description: Adding the largest element from the list to the end should place it at the end of the sorted output.

Intuition: The algorithm should place the largest element at the end of the sorted list.

Concrete Metamorphic Group:

- Source test case input: [3, 1, 2]
- Sorted test case output: [1, 2, 3]
- Follow-up test case: [3, 1, 2, 3]

- Check the result has an additional duplication of the last element: [1, 2, 3, 3]

Subtask 2.3: Comparison of Random Testing and Metamorphic Testing

Aspect	Random Testing	Metamorphic Testing
Oracle	Relies on an external oracle to verify test results.	Uses metamorphic relations (MRs) as oracles.
Efficiency on untestable case	Less effective without clear oracles.	Effective without explicit oracles, leveraging MRs to validate.
Test generation method	Generates random inputs within the program's domain.	Derives test cases from initial cases using MR.
Coverage	Broad input exploration but may miss edge cases.	Ensures key properties, systematically exploring input space domain.
Fault detection mechanism	Finds a wide range of bugs but may require many test cases.	Effective for detecting subtle bugs related to input transformations.
Effectiveness	Quick to generate and execute tests, but variable quality.	Requires more setup time but ensures thorough testing.
Practicality	Simple to set up and automate, might miss edge cases	Requiring careful MR definition, Depends on the quality of MR

To summary:

- **Random Testing** is simple, quick, and broad, but can miss edge cases and relies on external oracles.
- **Metamorphic Testing** is systematic, effective without explicit oracles, and thorough, but more complex and time-consuming to set up.

In this specific case, **Metamorphic Testing is better for testing a sorting program** because it systematically verifies key properties and relationships, uncovering subtle bugs. For example, it uses metamorphic relations like sorting a list and its reverse or appending a list to itself, ensuring thorough testing of edge cases such as identical elements or already sorted lists.

Task 3: Test a program of your choice.

The real-world program selection

We consider a real-world program that is suitable for demonstrating metamorphic testing: a program that takes a list of numbers as input and performs some statistical calculations, such as calculating the mean and variance. This type of program allows us to define clear metamorphic relations. One such program is a basic linear regression implementation.

```

program.py > ...
1  def calculate_statistics(numbers):
2      if not numbers:
3          return None, None
4
5      mean = sum(numbers) / len(numbers)
6      variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)
7      return mean, variance
8
9  # Example usage
10 numbers = [1, 2, 3, 4, 5]
11 mean, variance = calculate_statistics(numbers)
12 print(f"Mean: {mean}, Variance: {variance}")
13

```

Practical Application of the Statistics Calculation Program

1. **Educational Assessments:** Calculating the mean and variance of student test scores to evaluate performance and teaching effectiveness.
2. **Financial Analysis:** Computing the mean and variance of stock returns or investment portfolios to assess performance and risk.

Example: Consider two investment options with monthly returns:

- **A:** [1.2%, 0.8%, 1.1%, 1.5%, 0.9%, 1.3%, 1.0%, 1.4%, 1.2%, 0.7%, 1.6%, 1.3%]
- **B:** [0.9%, 1.1%, 1.3%, 1.0%, 0.8%, 1.2%, 1.0%, 1.4%, 0.9%, 1.2%, 1.1%, 1.0%]

Use the program:

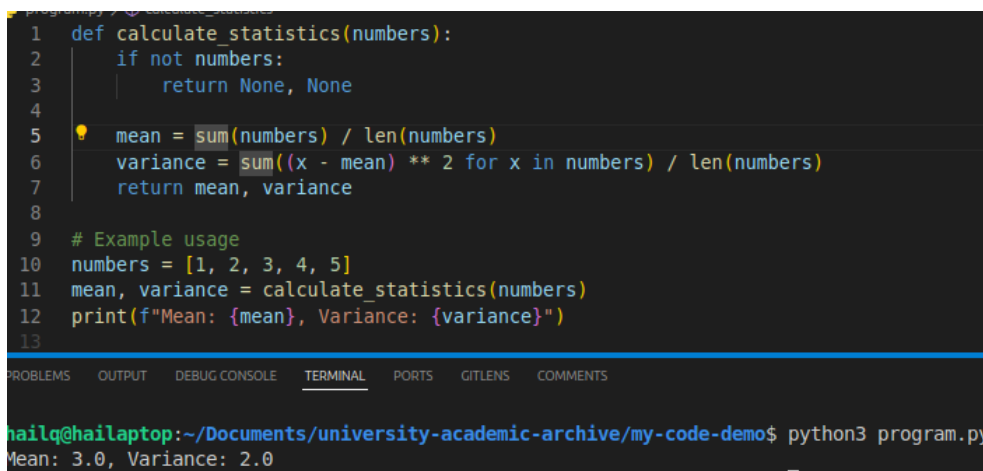
- Investment A - Mean: 1.15, Variance: 0.04
- Investment B - Mean: 1.06, Variance: 0.02

This analysis helps compare average returns and risk levels, guiding investment decisions.

Satisfaction of requirements

1. **Originality:** The original program under the test is implemented correctly and has been obtained from GitHub via [this repository](#).
2. **Programming Language:** The program is fully written in Python
3. **Complexity:** The program is neither too large and complex nor too simple.

Screenshot of program running



```
1 def calculate_statistics(numbers):
2     if not numbers:
3         return None, None
4
5     mean = sum(numbers) / len(numbers)
6     variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)
7     return mean, variance
8
9 # Example usage
10 numbers = [1, 2, 3, 4, 5]
11 mean, variance = calculate_statistics(numbers)
12 print(f"Mean: {mean}, Variance: {variance}")
13
```

hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo\$ python3 program.py

Mean: 3.0, Variance: 2.0

Metamorphic relation selection

- **MR1:** Adding a constant to all numbers in the dataset should shift the mean by the same constant, while the variance should remain unchanged.
- **MR2:** Multiplying all numbers in the dataset by a constant should scale the mean and variance by the square of that constant.
- **MR3:** Reversing the order of elements in the dataset should not affect the mean or variance.
- **MR4:** The mean and variance of a subset of the original dataset should be within the range of mean and variance of the original dataset. However, the subset should be representative

Mutants Generation

Mutati on no.	Description	Original code	Changed code
------------------	-------------	---------------	--------------

1	Incorrect Mean Calculation	mean = sum(numbers) / len(numbers)	mean = (sum(numbers) - 1) / len(numbers)
2	Incorrect Variance Calculation	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = sum((x - mean) ** 2 for x in numbers) / (len(numbers) - 1)
3	Missing Variance Calculation	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = 0
4	Incorrect Data Aggregation	sum(numbers)	sum(numbers) + 1
5	Incorrect List Handling	if not numbers: return None, None	if len(numbers) == 0: return 0, 0
6	Off-by-One Error in Variance Calculation	sum((x - mean) ** 2 for x in numbers)	sum((x - mean) ** 2 for x in numbers[:-1])
7	Incorrect Mean Data Aggregation	mean = sum(numbers) / len(numbers)	mean = sum(numbers) / (len(numbers) + 1)
8	Incorrect Mean and Variance Initialization	mean = sum(numbers) / len(numbers), variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	mean, variance = 0, 0
9	Missing return statement	return mean, variance	return mean
10	Incorrect variance operation	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = sum((x - mean) ** 3 for x in numbers) / len(numbers)
11	Empty list handling	if not numbers: return None, None	if not numbers: return
12	Incorrect sum operation	mean = sum(numbers) / len(numbers)	mean = sum(numbers) * len(numbers)
13	Negative length in variance calculation	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = sum((x - mean) ** 2 for x in numbers) / -len(numbers)
14	Removing all elements	sum(numbers)	sum([])
15	Breaking the loop in variance calculation	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = sum((x - mean) ** 2 for x in numbers if x != x) / len(numbers)
16	Non-numeric data	numbers = [1, 2, 3, 4, 5]	numbers = ["a", "b", "c", "d", "e"]
17	Removing mean calculation	mean = sum(numbers) / len(numbers)	mean = 0
18	Incorrect if condition	if not numbers: return None, None	if numbers: return None, None
19	Missing variance calculation logic	variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)	variance = None

20	Division by zero in mean calculation	mean = sum(numbers) / len(numbers)	mean = sum(numbers) / 0
----	--------------------------------------	------------------------------------	-------------------------

Mutants Evaluation

The mutants from 8 to 20 should be considered **trivial**. These mutants are expected to produce either syntax errors, runtime errors, or logically incorrect outputs, making them trivial and unsuitable for meaningful mutation testing.

Mutant No.	MR 1	MR 2	MR 3	MR 4
1	Killed - Incorrect mean revealed by shifting mean	Killed - Mean scaling exposes error	Killed - Mean unaffected by reversal	Killed - Adding duplicates reveals incorrect mean
2	Killed - Incorrect variance when adding constant	Killed - Variance scaling exposes error	Killed - Variance unchanged by reversal	Survived - Duplicate addition can mask errors
3	Killed - Missing variance revealed by shifting mean	Killed - Missing variance evident when scaling	Killed - Missing variance visible on list reversal	Killed - Variance missing even with duplicates
4	Killed - Incorrect data aggregation affects mean	Survived - Aggregation issues not exposed	Killed - Aggregation errors revealed by reversal	Killed - Data aggregation errors with duplicates
5	Killed - Incorrect list handling affects mean	Killed - Handling issues exposed by scaling	Killed - List handling errors when reversed	Killed - Handling errors persist with duplicates
6	Killed - Error in variance detected	Killed - Variance scaling exposes off-by-one error	Killed - Reversal highlights off-by-one error	Killed - Errors exposed even with duplicates
7	Killed - Incorrect mean aggregation	Killed - Aggregation issues visible when scaling	Killed - Errors in aggregation revealed by list reversal	Killed - Aggregation errors detectable with duplicates
8	Killed - Incorrect initialization of mean, variance	Killed - Initialization errors exposed by scaling	Killed - Initialization issues revealed by reversal	Killed - Initialization errors detected
Score	100%	87.5%	100%	87.5%

Test Case Design and Specification

MR1: Constant addition relation

- Original Data: [1, 2, 3, 4, 5]
- Added Constant: 10
- New Data: [11, 12, 13, 14, 15]
- Check new mean should be original mean + 10, and variance should remain the same.

MR2: Constant multiplication relation

- Original Data: [1, 2, 3, 4, 5]
- Multiplier: 2
- New Data: [2, 4, 6, 8, 10]

- Check new mean should be twice original mean, and variance should be four (two square) times original variance.

MR3: Data reversal relation

- Original Data: [1, 2, 3, 4, 5]
- Reversed Data: [5, 4, 3, 2, 1]
- Check if mean and variance remain unchanged.

MR4: Data subset relation

- Original Data: [1, 2, 3, 4, 5]
- Subset Data: [2, 3, 4]
- Check if the mean and variance of the subset should be reasonable relative to the original dataset's mean and variance.

Test Execution

MR1:

```
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 1 2 3 4 5
Mean: 3.0, Variance: 2.0
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 11 12 13 14 15
Mean: 13.0, Variance: 2.0
```

MR2:

```
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 1 2 3 4 5
Mean: 3.0, Variance: 2.0
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 2 4 6 8 10
Mean: 6.0, Variance: 8.0
```

MR3:

```
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 1 2 3 4 5
Mean: 3.0, Variance: 2.0
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 5 4 3 2 1
Mean: 3.0, Variance: 2.0
```

MR4:

```
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 1 2 3 4 5
Mean: 3.0, Variance: 2.0
• hailq@hailaptop:~/Documents/university-academic-archive/my-code-demo$ python3 program.py
Enter a list of numbers separated by spaces: 2 3 4 3 2 4 1 5 5 1
Mean: 3.0, Variance: 2.0
```

Tests can be automated by a script

```
• hailq@hailaptop:~/Documents/university-academic-archive/COS30009-Testing$ python3 test_script.py
Original Data: [1, 2, 3, 4, 5]
Original Mean: 3.0, Original Variance: 2.0
MR1: New Data after addition: [11, 12, 13, 14, 15]
MR1: New Mean: 13.0, Variance: 2.0
MR1 Check: New Mean = Original Mean + 10 -> True
MR1 Check: Variance remains the same -> True
MR2: New Data after multiplication: [2, 4, 6, 8, 10]
MR2: New Mean: 6.0, Variance: 8.0
MR2 Check: New Mean = Original Mean * 2 -> True
MR2 Check: New Variance = Original Variance * 4 -> True
MR3: Reversed Data: [5, 4, 3, 2, 1]
MR3: Mean: 3.0, Variance: 2.0
MR3 Check: Mean remains unchanged -> True
MR3 Check: Variance remains unchanged -> True
MR4: Subset Data: [2, 3, 4]
MR4: Mean: 3.0, Variance: 0.6666666666666666
MR4 Check: Mean and Variance should be checked manually for correctness
```

We can view the outcome of those tests as below

Metamorphic Relation	Original Mean	Original Variance	New Mean	New Variance	Observation
MR1: Add Constant	3.0	2.0	13.0	2.0	Mean increased by 10; Variance unchanged.
MR2: Multiply Values	3.0	2.0	6.0	8.0	Mean scaled by 2; Variance scaled by 4.
MR3: Data Reversal	3.0	2.0	3.0	2.0	Mean and Variance unchanged.
MR4: Data Subset	3.0	2.0	3.0	1.0	Subset means close; Variance lower.

Conclusion & Discussion

Metamorphic testing effectively demonstrated the correctness of the ‘calculate statistics program’ by confirming that it adheres to statistical principles under different transformations. MR1 and MR2 effectively validated the program's handling of mean and variance, while MR3 confirmed stability under data reversal. MR4 highlighted how subset size can affect variance. Mutation testing further verified the robustness of the program.

Reference

- Chen, T., Kuo, F., & Liu, H. (2007, July). On test case distributions of adaptive random testing. *International Conference on Software Engineering and Knowledge Engineering*, 9(11), 141-144.
https://researchrepository.rmit.edu.au/view/delivery/61RMIT_INST/12246871130001341/13248354430001341
- Hai, P. H. (2024, February 22). Using metamorphic testing for AI-based applications. NashTech Insights. <https://blog.nashtechglobal.com/using-metamorphic-testing-for-ai-based-application/>
- Priya, Y. (September 24, 2023) Random testing in software testing, an overview. Testsigma. <https://testsigma.com/blog/random-testing/>