Swinburne University of Technology
Department of Computing Technologies
Software Testing and Reliability
SWE30009

Name: Nguyen Duc Le Nguyen

Id: 104224493

Report Project

Task 1: Random testing

Subtask 1.1

Random testing is a software testing technique where inputs are generated randomly within the input domain of the program under test. Random testing seeks to cover a broad scope of unexplored areas and in this case unexploding and excluding any out-of-band test that was not provided for by handwritten use cases. This means that using randomness in testing different aspects of the software one can find other bugs/ errors that would have been missed otherwise and the product will work as intended in all the input scenarios.

Example Illustration:

Imagine testing a calculator app. While we might manually test operations like 2+22 or 5×35, random testing might generate inputs like 123456789×987654321 or division by zero, revealing issues that manual tests might miss.

Distribution Profiles for Random Testing

The effectiveness of random testing often depends on the distribution profile used to generate inputs. There are two primary distribution profiles:

1. **Uniform Distribution:** In a uniform distribution, each possible input has an equal probability of being selected. This approach is beneficial for ensuring a comprehensive and fair exploration of the input space, as it doesn't favor any values. However, it may become inefficient if the input domain is vast, especially when some regions are known to be more error-prone than others.

2. **Non-Uniform (Biased) Distribution:** In a non-uniform distribution, inputs are generated based on a probability distribution that emphasizes certain values or ranges. This allows testers to concentrate on inputs more likely to expose defects, such as boundary values or those with a history of issues, providing a targeted approach to uncovering potential faults.

Example Illustration:

If testing a function that handles arrays, we might bias the distribution to generate more empty arrays or arrays with maximum allowed size, as these are common sources of errors.

Process of Random Testing

1. **Define the Input Domain:** Clearly specify the range and types of inputs the program accepts. (e.g. for a function accepting integers from 1 to 1000, the input domain is [1, 1000].)
2. **Choose a Distribution Profile:** Decide between uniform or non-uniform distribution based on testing goals. (e.g. Use a uniform distribution to cover the entire input range equally.)
3. **Generate Random Inputs**: Utilize random number generators or tools to produce inputs according to the chosen distribution.
4. **Execute the Program with Generated Inputs:** Run the program using random inputs.
5. **Monitor and Check Outputs:** Verify that the program's outputs are correct.
6. **Record and Analyze Results:** Log inputs and outputs for further analysis to identify any defects or unusual behaviors
7. **Repeat the Process:** Continue generating and testing inputs to thoroughly explore the input domain.

Applications: Random testing is applicable in various scenarios:

1. **Stress Testing:** Assess how software behaves under extreme conditions or high load.
2. **Fuzz Testing:** Identify security vulnerabilities by providing random, unexpected, or malformed data inputs.
3. **Regression Testing:** Ensure that new changes haven't introduced defects by randomly testing different parts of the software.
4. **Simulation and Modeling:** Test systems like games or simulations where unpredictability is inherent.

Subtask 1.2: Applying Random Testing to a Sorting Program

1. **Define the Input Domain**: The input domain consists of a list with a length ranging from **1 to 10**, where each element is an integer between **-100 and 100.**

2. **Choose a Distribution Profile**: Uniform Distribution

3. **Generate Random Inputs**

Test case	Randomly select list length	Input List	Expected Sorted List	Note
1	5	[23, -47, 23, 0, 89]	[-47, 0, 23, 23, 89]	Has duplicate
2	8	[-100, 50, -50, 50, 100, 0, 0, 75]	[-100, -50, 0, 0, 50, 50, 75, 100]	Has duplicate
3	4	[12, -34, 67, -34]	[-34, -34, 12, 67]	No duplicate
4	10	[45, -1, 89, 67, -22, 0, 45, -50, 23, -100]	[-100, -50, -22, -1, 0, 23, 45, 45, 67, 89]	Has duplicate
5	3	[100, -100, 0]	[-100, 0, 100]	No duplicate

Task 2: Metamorphic testing

Subtask 2.1

A test oracle is a mechanism or process that decides whether the outputs of a program for given inputs are correct. It acts like a benchmark against which the correctness of the software under test is validated. Traditionally, in testing, test oracles provide expected outputs for given inputs, and testers compare actual outputs with respect to expected ones.

Example Illustration:

For a function that calculates the square of a number, the test oracle would specify that for input n=3, the expected output is 9.

Untestable Systems: is a system for which creating a test oracle is challenging or impossible. This may occur due to:

- **Complex Calculations or Algorithms:** Systems with complex numerical computations, like scientific simulations or machine learning, often have outputs that are difficult to predict precisely due to their mathematical complexity.
- **Non-Deterministic Behavior:** Systems with random or probabilistic elements, such as Monte Carlo simulations, produce varying results with each run, making outcomes unpredictable.
- **Lack of Specifications:** In projects with incomplete requirements, like legacy systems with poor documentation, predicting behavior is challenging due to unclear or missing specifications.
- **Huge Input Domain:** Systems with large input spaces, like compilers handling diverse code snippets, can't realistically define outputs for every possible input.

Example illustration: Testing a weather prediction model that takes huge amounts of data and uses complex algorithms to predict weather patterns. It's impractical to define the exact expected output for every possible input scenario.

Metamorphic Testing addresses the challenge of *untestable systems* by introducing a novel way to verify program behavior:

- Instead of focusing on individual inputs and outputs, MT examines relationships between multiple inputs and their corresponding outputs, called *metamorphic relations (MRs)*.
- These MRs define how the output should change when the input is modified in a specific way.

Intuition: If a system behaves correctly, modifying the input according to an MR should result in predictable changes in the output.

A **Metamorphic Relations** is a property or relation that defines how changes in the input should correspond to changes in the output. MRs are the cornerstone of MT.

Examples of MRs:

- For a sorting algorithm, reversing the input list and re-sorting should yield the same result as the initial sort.
- For a machine learning classifier, augmenting input data (e.g., flipping an image) should not change the predicted class for certain cases.

Process of Metamorphic Testing

1. Identify MRs: Analyze the program's domain and requirements to define MRs.
2. Create Test Inputs: Design initial test cases.
3. Apply Transformations: Modify the inputs according to the identified MRs.
4. Compare Outputs: Check if the outputs of the transformed inputs adhere to the MRs.

Some Applications of metamorphic testing

- **Scientific Computing:** Validation of numerical simulations.
- **Machine Learning:** Ensuring consistency of predictions.
- **Graph Algorithms:** Testing properties like symmetry or connectivity.
- **Sorting Algorithms:** Verifying sorted order for modified inputs.

*Illustration Example***:** A temperature conversion program (Celsius to Fahrenheit):

- MR: Adding 10°C to the input should result in an output increment of 18°F (based on the conversion formula).

Subtask 2.2

MR1: Reversal of Input

Description

Reversing the order of elements in the input list should not affect the sorted output. The sorting algorithm should produce the same sorted list regardless of the order of the input.

Intuition

Sorting is invariant to the arrangement of the input list. A valid sorting algorithm focuses on the values of the elements, not their initial positions. Thus, reversing the input should yield the same sorted result.

How It Works

1. Take an initial input list.
2. Reverse the order of the elements in the input list.
3. Sort both the original list and the reversed list using the program.

4. Compare the two outputs to verify they are identical.

Concrete Metamorphic Group

Name	Initial Input	Transformed Input	Expected Output
Small List	[5, 2]	[2, 5]	[2, 5]
List with Duplicates	[4, 3, 4, 2]	[2, 4, 3, 4]	[2, 3, 4, 4]
Already Sorted List	[1, 2, 3]	[3, 2, 1]	[1, 2, 3]
Single-Element List	[7]	[7]	[7]

MR2: Addition of Duplicates

Description

Adding duplicate elements to the input list should result in the same output as the original, with the added elements appearing at appropriate positions in the sorted list.

Intuition

Sorting considers the frequency of each element. Adding duplicates should only increase the count of the duplicated numbers in the sorted list, while maintaining the relative order of the elements.

How It Works

1. Take an initial input list.
2. Add duplicates of one or more elements in the input list.
3. Sort both the original list and the transformed list using the program.
4. Verify that the transformed list's output includes the original sorted result with the duplicates positioned correctly.

Concrete Metamorphic Group

Name	Initial Input	Transformed Input	Original Expected Output	Transformed Expected Output
Single Duplicate	[3, 1, 2]	[3, 1, 2, 2]	[1, 2, 3]	[1, 2, 2, 3]
Multiple Duplicates	[4, 5, 3]	[4, 5, 3, 4, 5]	[3, 4, 5]	[3, 4, 4, 5, 5]
All Elements Duplicated	[2, 6, 1]	[2, 6, 1, 2, 6, 1]	[1, 2, 6]	[1, 1, 2, 2, 6, 6]
Duplicate Existing Duplicates	[2, 2, 3]	[2, 2, 3, 2]	[2, 2, 3]	[2, 2, 2, 3]

Subtask 2.3

Aspect	Random Testing	Metamorphic Testing
Oracle Dependency	Requires a reliable oracle to verify correctness.	Does not require an oracle; relies on MRs.

Test Case Generation	Generates input cases randomly.	Systematically transforms inputs based on MRs.
Effectiveness	Often effective at finding superficial bugs.	Effective for untestable systems and complex domains.
Cost	Lower cost for test generation but may miss bugs.	Higher cost in defining MRs but detects deeper issues.

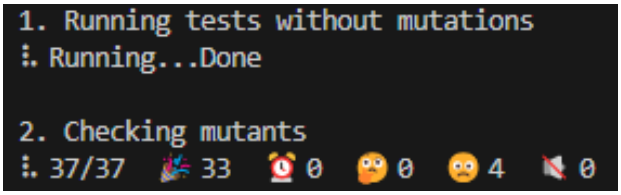
Task 3: Report on Metamorphic Testing for Prime Number Function

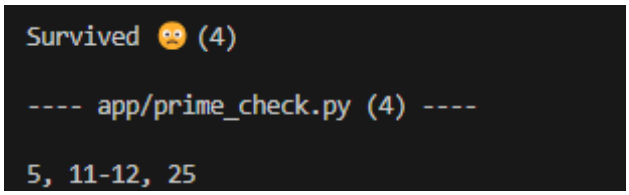
Objective

To test the is_prime function using metamorphic testing principles by evaluating its resilience against 37 generated mutants.

Mutant Overview

Total Mutants Generated: 37 **Mutants Killed**: 33 **Mutants Survived**: 4





Note: Mutant ids 11,12,14,21 are equivalent mutants

Mutant id	Original	Mutant	Status
1	if not isinstance(number, int) or not number >= 0:	if isinstance(number, int) or not number >= 0:	Killed
2		if not isinstance(number, int) or number >= 0:	Killed
3		if not isinstance(number, int) or not number > 0:	Killed
4		if not isinstance(number, int) or not number >= 1:	Killed
5		if not isinstance(number, int) and not number >= 0:	Survived
6	if 1 < number < 4:	if 2 < number < 4:	Killed
7		if 1 <= number < 4:	Killed
8		if 1 < number <= 4:	Killed
9		if 1 < number < 5:	Killed
10	if 1 < number < 4: return True	return False	Killed
~~11~~	elif number < 2 or number % 2 == 0 or number % 3 == 0:	~~elif number <= 2 or number % 2 == 0 or number % 3 == 0:~~	Survived
~~12~~		~~elif number < 3 or number % 2 == 0 or number % 3 == 0:~~	Survived
13		elif number < 2 or number / 2 == 0 or number % 3 == 0:	Killed
~~14~~		~~elif number < 2 or number % 3 == 0 or number % 3 == 0:~~	Killed
15		elif number < 2 or number % 2 != 0 or number % 3 == 0:	Killed
16		elif number < 2 or number % 2 == 1 or number % 3 == 0:	Killed
17		elif number < 2 or number % 2 == 0 or number / 3 == 0:	Killed
18		elif number < 2 or number % 2 == 0 or number % 4 == 0:	Killed
19		elif number < 2 or number % 2 == 0 or number % 3 != 0:	Killed
20		elif number < 2 or number % 2 == 0 or number % 3 == 1:	Killed

~~21~~		~~elif number < 2 and number % 2 == 0 or number % 3 == 0:~~	**Killed**
22	elif number < 2 or number % 2 == 0 or number % 3 == 0: return False	return True	**Killed**
23	for i in range(5, int(math.sqrt(number) + 1), 6):	for i in range(6, int(math.sqrt(number) + 1), 6):	**Killed**
24		or i in range(5, int(math.sqrt(number) - 1), 6):	**Killed**
25		for i in range(5, int(math.sqrt(number) + 2), 6):	**Survived**
26		for i in range(5, int(math.sqrt(number) + 1), 7):	**Killed**
27	if number % i == 0 or number % (i + 2) == 0:	if number / i == 0 or number % (i + 2) == 0:	**Killed**
28		if number % i != 0 or number % (i + 2) == 0:	**Killed**
29		if number % i == 1 or number % (i + 2) == 0:	**Killed**
30		if number % i == 0 or number / (i + 2) == 0:	**Killed**
31		if number % i == 0 or number % (i - 2) == 0:	**Killed**
32		if number % i == 0 or number % (i + 3) == 0:	**Killed**
33		if number % i == 0 or number % (i + 2) != 0:	**Killed**
34		if number % i == 0 or number % (i + 2) == 1:	**Killed**
35		if number % i == 0 and number % (i + 2) == 0:	**Killed**

36	for i in range(5, int(math.sqrt(number) + 1), 6):
	if number % i == 0 or number % (i + 2) == 0:
	return False
	return True
		return True
		return True
			Killed

37		return False return False	**Killed**

Testing Overview:

1. Zero and One Metamorphic Relation

```python
def test_zero_and_one_metamorphic():

    assert is_prime(0) is False
    assert is_prime(1) is False
    # Metamorphic transformations and expected results
    transformations = [
        lambda x: x + 0,  # Adding zero
        lambda x: x * 1,  # Multiplying by one
        lambda x: x - 0,]  # Subtracting zero
    # Test metamorphic transformations on 0
    for transform in transformations:
        assert is_prime(transform(0)) is False
    # Test metamorphic transformations on 1
    for transform in transformations:
        assert is_prime(transform(1)) is False
```

```
2. Checking mutants
⠋ 37/37  🎉 7  ⏰ 0  🤔 0  🙁 30  🔇 0
PS D:\SWB\SWE30009\Task3> python -m mutmut show
To apply a mutant on disk:
    mutmut apply <id>

To show a mutant:
    mutmut show <id>


Survived 🙁 (30)

---- app/prime_check.py (30) ----

5-6, 8-20, 23-37
```

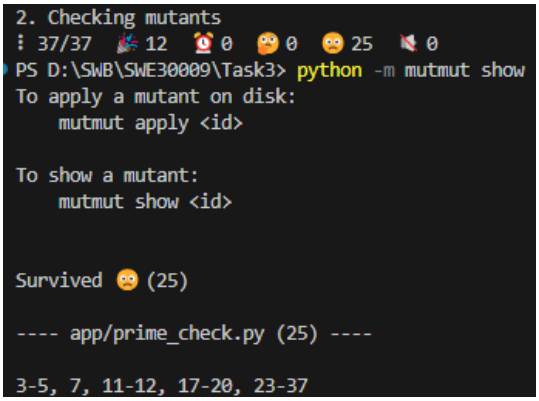
Description: Tests edge cases where 0 and 1 are non-prime by applying transformations like: Adding 0 (x + 0), Multiplying by 1 (x * 1), Subtracting 0 (x - 0).

Key Mutants Killed: 1, 2, 3, 4, 7, 21, 22 (7 mutants)

Reason for Effectiveness: These transformations are identity operations, ensuring no logical change occurs in input. Any mutant altering this logic (e.g., incorrectly treating 0 or 1 as prime) was successfully identified.

2. Non-Prime Powers of Two Metamorphic Relation

```python
def test_non_prime_powers_of_two_metamorphic()
    # Test powers of two: 2^n (for n >= 1)
    for n in range(1, 10):
        power_of_two = 2 ** n
        if n == 1:  # Special case: 2^1 = 2 is prime
            assert is_prime(power_of_two) is True
        else:
            assert is_prime(power_of_two) is False
```



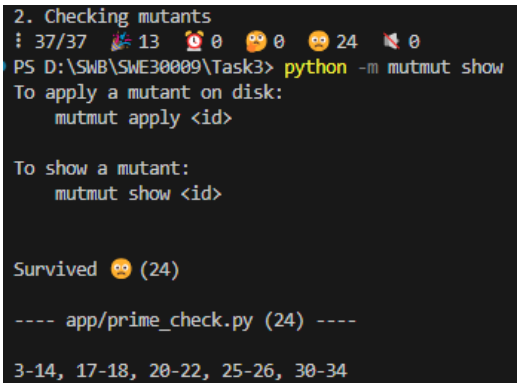
Description: Tests powers of 2 (2^n for n >= 1) to ensure: Only 2 is prime and all other powers of 2 are non-prime.

Key Mutants Killed: 1, 2, 6, 8, 9, 10, 12, 14, 15, 16, 21, 22 (12 mutants)

Reason for Effectiveness: Mutants that fail to distinguish between the primality of 2 and other powers of 2 were caught. These cases are critical for ensuring correctness in even-number handling.

3. Prime Squared Metamorphic Relation

```python
def test_metamorphic_property_prime_squared():
    # The square of any prime number should not be a prime number
    prime = 5
    assert is_prime(prime) is True
    assert is_prime(prime ** 2) is False
    # 25 is not prime
```



Description: Verifies that the square of any prime number is non-prime.

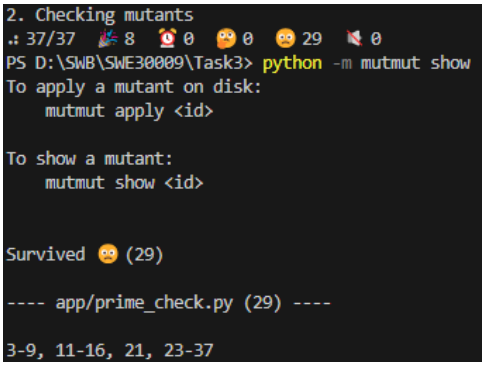
Key Mutants Killed: 1, 2, 15, 16, 19, 23, 24, 27, 28, 29, 35, 36, 37 (13 mutants)

Reason for Effectiveness: Squaring a prime produces a composite number, and mutants incorrectly classifying squared primes as prime were killed.

4. Composite Minus Prime Metamorphic Relation

```python
def test_metamorphic_property_composite_minus_prime():

    composite = 15
    prime = 3
    assert is_prime(composite) is False
    assert is_prime(prime) is True
    assert is_prime(composite - prime) is False
```



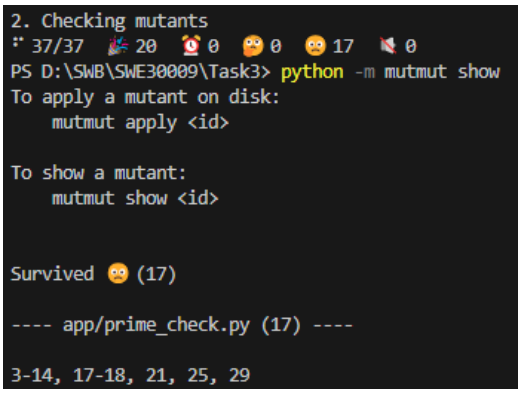
Description: Validates subtracting a prime from a composite number of results in composite number or occasionally, another prime.

Key Mutants Killed: 1, 2, 10, 17, 18, 19, 20, 22 (8 mutants)

Reason for Effectiveness: Mutants misclassifying results of composite-prime subtractions (e.g., 15 - 3 = 12, a composite) as prime or misidentifying the primality of intermediate results were identified.

5. 6k ± 1 Rule Metamorphic Relation

```python
def test_6k_metamorphic():
    base_number = 120  # composite 121
    sqrt_of_next_prime = 127  # prime over sqrt(121)
    assert is_prime(base_number) is False  # 6*20 = 120, is non-prime
    assert is_prime(base_number - 1) is False  # 6*20 - 1 = 119, is non-prime
    assert is_prime(base_number + 1) is False  # 6*20 + 1 = 121, is non-prime
    assert is_prime(sqrt_of_next_prime) is True  # 127 is prime
    assert is_prime(sqrt_of_next_prime - 6) is False  # 127 - 6 = 121, is non-prime
```



Description: Tests numbers fitting the form 6k ± 1, which are potential primes, as well as composite numbers around this form

Key Mutants Killed: 1, 2, 15, 16, 19, 20, 22, 23, 24, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 37 (20 mutants)

Reason for Effectiveness: The 6k ± 1 rule efficiently identifies non-primes like 6k6k6k, 6k−16k - 16k−1, and 6k+16k + 16k+1 while ensuring correctness for actual primes fitting the rule.

Effectiveness Summary

Metamorphic Relation	Mutants Killed	Primary Focus
Zero and One Metamorphic	7	Validates edge cases for 0 and 1.
Non-Prime Powers of Two Metamorphic	12	Ensures correct classification of even numbers.
Prime Squared	13	Verifies squares of primes are non-prime.

Composite Minus Prime		Tests primality of results from composite-prime subtractions.
6k ± 1 Rule Metamorphic	20	Tests pattern-based primality for numbers in the 6k±1 form.
Total killed	33/37 Mutants	

Improvements and Insights

Strengths of the Testing Process

- **High Coverage:** The test suite killed 33 out of 37 mutants, demonstrating its effectiveness in identifying logical flaws. In addition, the **6k ± 1 Rule MR** was the most effective, killing 20 mutants due to its broad applicability.
- **Systematic Validation:** The metamorphic relations ensured comprehensive testing across diverse scenarios, including edge cases, patterns, and boundary conditions.

Weaknesses and Gaps

- **Surviving Mutants:** Four mutants survived due to insufficient edge case coverage or lack of invalid input handling.
- **Range Limitations:** The current test cases did not include very large numbers, limiting the detection of potential overflow or performance issues.

Improvements

- **Edge Case Tests:** Add tests for negative numbers, floating-point inputs, and non-integer values.
- **Boundary Tests:** Explicitly test numbers like 2 and 3 to address mutants altering range checks.
- **Performance Tests:** Evaluate the function for large primes and composites to ensure computational efficiency.
- **Expanding Metamorphic Relations:** Add additional transformations to handle divisors near square roots and extended ranges.

Conclusion

The metamorphic testing approach successfully validated the is_prime function, achieving an 89.2% mutation score. While the current test suite demonstrates robustness, addressing the identified gaps can further enhance its effectiveness, ensuring complete resilience against logical flaws.