SWE30009

Sofware Testing and Reliability

Project Report

Student Name: Lau Ngoc Quyen

Student ID: 104198996

Task 1: Random Testing

Subtask 1.1: Understanding Random Testing

Intuition of Random Testing:

Random testing is a method where test cases are generated randomly, aiming to explore a program's input space without bias. This approach helps identify defects not found with structured methods, increasing the likelihood of encountering unexpected behavior or edge cases.

Distribution Profiles for Random Testing:

- **Uniform Distribution:** Each possible value in the input space has an equal probability of being selected. This is the simplest form of random testing.
- **Biased Distribution:** Certain values are given higher probabilities based on specific criteria or knowledge about the system. For example, boundary values or common user inputs might be tested more frequently.

Process of Random Testing:

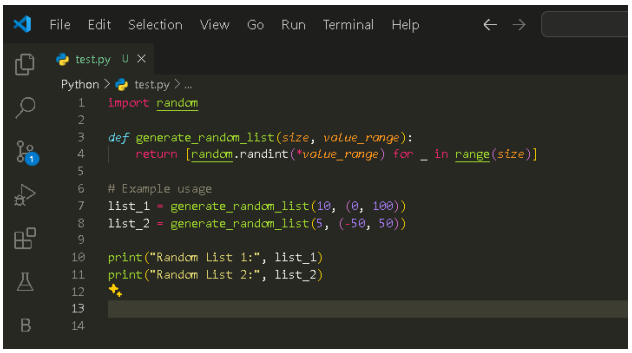
1. **Define Input Domain:** Identify the range and type of inputs the program can accept.
2. **Generate Random Inputs:** Use a random generator to create input values within the defined domain.
3. **Execute Test Cases:** Run the program with the randomly generated inputs.
4. **Check Results:** Compare the program's output with the expected output (if known) or check for crashes and exceptions.

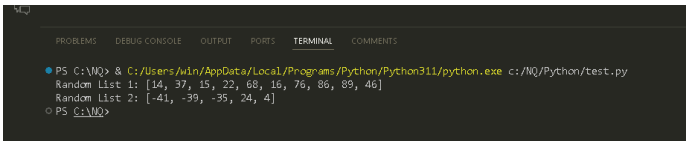
Applications of Random Testing:

- **Stress Testing:** Evaluating how a system performs under extreme conditions by generating a large number of random inputs.
- **Regression Testing:** Ensuring that new changes have not introduced new bugs by running random tests.
- **Security Testing:** Identifying vulnerabilities by generating unexpected or malicious inputs.

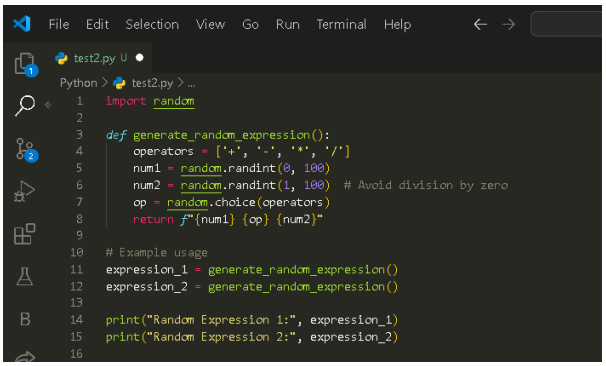
Examples/Illustrations:

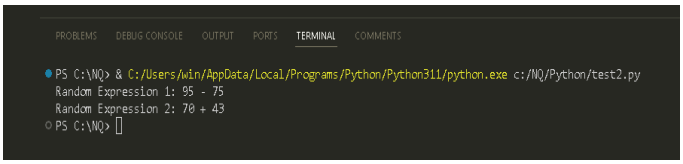
- **Example 1:** Test a sorting algorithm by generating random lists of integers and checking if the output is correctly sorted.





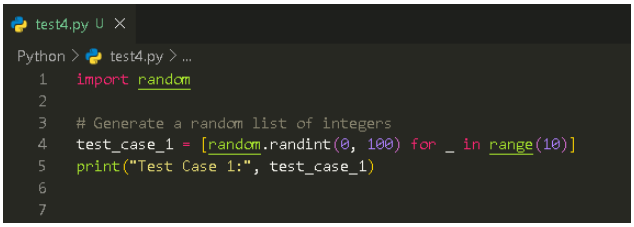
- **Example 2:** Test a basic calculator program by generating random arithmetic expressions and verifying correctness.





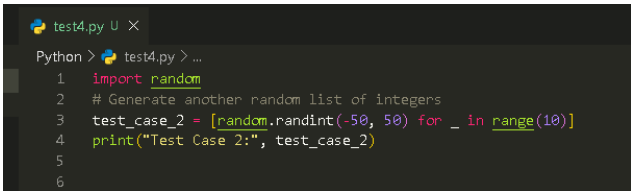
Subtask 1.2: Applying Random Testing

Test Case 1: Generate a random list of integers



Output:

Test Case 2: Generate another random list of integers



Output:

Task 2: Metamorphic Testing

Subtask 2.1: Understanding Metamorphic Testing

Test Oracle and Untestable Systems:

- **Test Oracle:** A mechanism for determining whether the outcomes of test cases are correct. In many cases, it's challenging to establish a reliable test oracle due to the complexity of the system under test.
- **Untestable Systems:** Systems for which it is difficult or impossible to determine the correctness of test outcomes using traditional methods, often due to the absence of a test oracle or the complexity of expected outcomes.

Motivation and Intuition of Metamorphic Testing:

Metamorphic testing is motivated by the need to test systems that are difficult to test using conventional methods. It leverages the relationships between multiple inputs and their corresponding outputs to create additional test cases. The intuition is that even if we don't know the correct output for a single input, we can use known relationships (metamorphic relations) to derive additional test cases and check the consistency of outputs.

Metamorphic Relations:

Metamorphic relations (MRs) are properties that specify how the output should change when the input is modified in specific ways. For example, in a sorting program, if an input list is reversed, the output should also be reversed.

Process of Metamorphic Testing:

1. **Identify Metamorphic Relations:** Determine the relationships that should hold between inputs and outputs.
2. **Generate Test Cases:** Create initial test cases.
3. **Apply MRs:** Modify the inputs based on the identified MRs to create follow-up test cases.
4. **Execute Test Cases:** Run the program with both the original and follow-up test cases.
5. **Check Consistency:** Verify if the outputs of the follow-up test cases adhere to the expected relationships defined by the MRs.

Applications of Metamorphic Testing:

- **Machine Learning Systems:** Where exact outputs are difficult to predict.
- **Simulations and Modelling:** Where precise outcomes are complex and varied.
- **Data Processing Pipelines:** Where transformations need to maintain certain properties.

Subtask 2.2: Applying Metamorphic Testing:

Example MR 1: Reversing a Sorted List

Source Input (SI): [2, 1, 3, 5, 4]

Source Output (SO): [1, 2, 3, 4, 5]

Metamorphic Relation 1 (MR1): Reversing a Sorted List
Follow-up Input (FI): [4, 3, 5, 2, 1]
Follow-up Ouput (FO): [1, 2, 3, 4, 5]

Example MR 2: Adding an Element to a Sorted List

Source Input (SI): [2, 1, 3, 5, 4]

Source Output (SO): [1, 2, 3, 4, 5]

Metamorphic Relation 1 (MR1): Reversing a Sorted List
Follow-up Input (FI): [4, 3, 5, 2, 1, 6]
Follow-up Ouput (FO): [1, 2, 3, 4, 5, 6]

Subtask 2.3: Comparing Random and Metamorphic Testing

Advantages and Disadvantages:

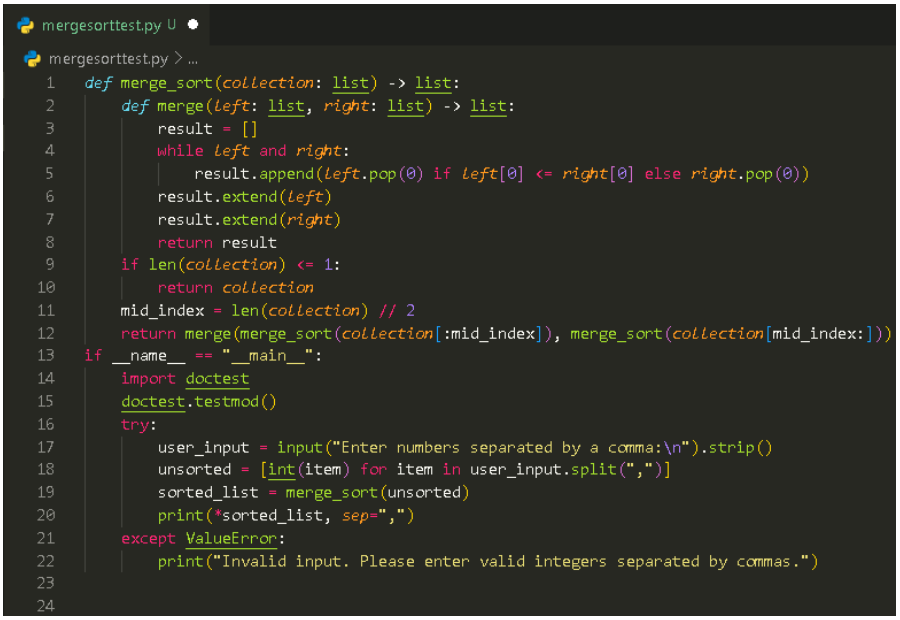
Aspect	Random Testing	Metamorphic Testing
Test Oracle	Requires a reliable test oracle for validation	Relies on metamorphic relations to verify consistency
Test Case Generation	Simple to generate large number of test cases	Requires identification and application of MRs
Effectiveness	Good at uncovering unexpected or rare defects	Effective for systems with complex or unknown outputs
Coverage	May lack thorough coverage if random inputs are not diverse	Provides more systematic coverage through MRs
Implementation	Relatively straightforward to implement	Requires understanding and defining MRs

Task 3: Testing a program of your choice

For this task, I will use a simple Python program that implements a sorting algorithm. The chosen program is a Python implementation of the Merge Sort algorithm, obtained from GitHub.

Link to code: https://github.com/TheAlgorithms/Python/blob/master/sorts/merge_sort.py

Testing Program: (Appendix A)



Reason for Selection:

- The sorting algorithm is simple enough to be manageable within the constraints of the project.
- It is complex enough to generate a sufficient number of mutants for mutation analysis.

Real-World Applications of Merge Sort Algorithm

- **Database Management Systems**: Merge sort is often used in databases for sorting large amounts of data, especially when the data cannot fit into memory all at once. It's efficient for external sorting algorithms, where data is sorted in chunks that are then merged together.

- **File Systems**: When managing files or directories, file systems might use merge sort to keep data organized. This can be useful for operations like listing files in a directory in sorted order.

- **Data Processing**: In big data processing frameworks like Hadoop and Spark, merge sort can be used to handle large-scale data sorting tasks. It's useful in scenarios where data is distributed across multiple machines and needs to be merged.

- **Search Algorithms**: Merge sort is used in some search algorithms and systems that need to maintain sorted order, such as in some types of search engines or indexing systems.

- **Networking**: In network protocols or systems that need to handle data streams and sort packets, merge sort can be used to organize and process data efficiently.

- **Scientific Computing**: Merge sort is applied in scientific computations that require sorting large datasets, such as in simulations or analysis of experimental data.

Applying Metamorphic Testing

Metamorphic Relation 1: Sorted List Relation

- **MR Description:** If you apply the merge sort algorithm to a list, and then apply it again to the result, the result should be the same as the first sorted result.
- **Concrete Example:**
  - Original Input: [3, 1, 2]
  - First Sort Result: [1, 2, 3]
  - Second Sort Result: [1, 2, 3]

Metamorphic Relation 2: Invariant Under Permutations

- **MR Description:** If you permute (shuffle) a sorted list and then sort it again, the result should be the same as the original sorted list.
- **Concrete Example:**
  - Original Input: [1, 2, 3]
  - Permuted Input: [2, 3, 1]
  - Sorted Permuted Input: [1, 2, 3]

Test Cases For Each Metamorphic Relation:

Test Case for MR 1:

Source Input (SI): [4, 2, 7, 1, 3]

Source Output (SO): [1, 2, 3, 4, 7]

Follow-up Input (FI): [1, 2, 3, 4, 7]

Follow-up Ouput (FO): [1, 2, 3, 4, 7]

Test Case for MR 2:

Source Input (SI): [5, 9, 2]

Source Output (SO): [2, 5, 9]

Follow-up Input (FI): [9, 5, 2]

Follow-up Ouput (FO): [2, 5, 9]

Mutants

Original Code: (Appendix A)

Mutant 1: Change **<=** to **<** in the merge function.

```
result.append(left.pop(0) if left[0] < right[0] else right.pop(0)) # Mutant 1
```

Mutant 2: Remove the **result.extend(left)** line.

```
# result.extend(left)  # Mutant 2
```

Mutant 3: Change the mid index calculation to **(len(collection) + 1) // 2.**

```
mid_index = len((collection)+1) // 2 # Mutant 3
```

Mutant 4: Change **<=** to **>** in the merge function.

```
result.append(left.pop(0) if left[0] > right[0] else right.pop(0)) # Mutant 4
```

Mutant 5: Reverse the order of merging in the merge function.

```
result.extend(right)
result.extend(left)  # Mutant 5
```

Mutant 6: Change **result.append(left.pop(0))** to **result.append(right.pop(0)).**

```
    result.append(right.pop(0))  # Mutant 6
```

Mutant 7: Remove **result.extend(right)** line in the merge function.

```
result.extend(left)  # Mutant 7: Removed result.extend(right)
```

Mutant 8: Add an **extra result.append(left.pop(0))** in the merge function.

```
result.append(left.pop(0))  # Mutant 8
result.extend(left)
result.extend(right)
```

Mutant 9: Change **if len(collection) <= 1:** to **if len(collection) < 1:**

```
if len(collection) < 1:  # Mutant 9
```

Mutant 10: Change **if len(collection) <= 1:** to **if len(collection) < 1:**

```
return merge(merge_sort(collection[mid_index:]), merge(merge_sort(collection[:mid_index])))  # Mutant 10
```

Mutant 11: Change **pop(0)** to **pop()**

```
result.append(left.pop() if left[0] <= right[0] else right.pop())  # Mutant 11
```

Mutant 12: Add an extra **result.extend(left)** after merging

```
result.extend(left)  # Mutant 12
result.extend(right)
result.extend(left)  # Mutant 12: Added extra result.extend(left)
```

Mutant 13: Change **result.extend(left)** to **result = left + result**

```
result.extend(left)  # Mutant 13
result.extend(right)
result = left + result  # Mutant 13: Changed to result = left + result
```

Mutant 14: Use **result.append(left.pop(0))** and **result.append(right.pop(0))** alternately

```
    result.append(right.pop(0))  # Mutant 14
```

Mutant 15: Remove **result = []** initialization

```
    # result = []  # Mutant 15: Removed
```

Mutant 16: Change **result.append(left.pop(0))** to r**esult.append(left.pop())**

```
result.append(left.pop() if left[0] <= right[0] else right.pop(0))  # Mutant 16
```

Mutant 17: Change **merge_sort(collection[:mid_index])** to **merge_sort(collection[mid:])**

```
left = merge_sort(collection[mid_index:])  # Mutant 17
```

Mutant 18: Change **merge_sort(collection[mid:])** to **merge_sort(collection[:mid_index])**

```
left = merge_sort(collection[:mid_index])  # Mutant 18
```

Mutant 19: Swap **left** and **right** in the merge function

```
result.append(right.pop(0) if left[0] <= right[0] else left.pop(0))  # Mutant 19
```

Mutant 20: Change **merge(left, right)** to **merge(left, right[::-1])**

```
return merge(merge_sort(collection[:mid_index]), merge_sort(collection[mid:][::-1]))  # Mutant 20
```

Supporting Table

	Test Case 1	Test Case 2	Killed
Original	[1, 2, 3, 4, 7]	[5, 9, 2]	No
Mutant 2	[4, 2, 7, 1, 3] → [1, 2, 3]	[5, 9, 2] → [2]	Yes
Mutant 4	[4, 2, 7, 1, 3] → [7, 4, 3, 2, 1]	[5, 9, 2] → [9, 5, 2]	Yes
Mutant 6	[4, 2, 7, 1, 3] → error	[5, 9, 2] → error	Yes
Mutant 8	[1, 2, 3, 4, 7] → error	[5, 9, 2] → error	Yes
Mutant 13	[4, 2, 7, 1, 3] → [4,2,4,7,1,3,7]	[5, 9, 2] → [5, 9, 2, 9]	Yes
Mutant 19	[4, 2, 7, 1, 3] → [7, 4, 3, 2, 1]	[5, 9, 2] → [9, 5, 2]	Yes
Mutant 20	[1, 2, 3, 4, 7] → error	[5, 9, 2] → error	Yes

Discussion:

- **Mutant 2**: Removing **result.extend(left)** resulted in missing elements from the **left** list in the final output. This omission caused incorrect results, leading to a failure of the test cases. Mutant 2 was effectively killed by the test cases.

- **Mutant 4**: Changing the comparison operator from <= to > in the merge function led to incorrect merging and sorting. This mutant produced incorrect results that did not match the expected sorted order, causing the test cases to fail. Mutant 4 was effectively killed.

- **Mutant 6**: Changing **result.append(left.pop(0))** to **result.append(right.pop(0))** led to errors during merging, as the wrong elements were appended. This change disrupted the correct sorting process, causing the test cases to fail. Mutant 6 was effectively killed.

- **Mutant 8**: Adding an extra **result.append(left.pop(0))** introduced additional elements into the result from the **left** list, causing incorrect output. This mutant was effectively killed by the test cases due to this erroneous result.

- **Mutant 13**: Changing **result.extend(left)** to **result = left + result** led to incorrect final output, as the merging of lists was disrupted. This mutant was effectively killed by the test cases because the results did not match the expected sorted order.

- **Mutant 19**: Swapping **left** and **right** in the **merge** function resulted in incorrect merging behavior. This mutation caused the output to be incorrect, failing the test cases. Mutant 19 was effectively killed.

- **Mutant 20**: Changing **merge(left, right)** to **merge(left, right[::-1])** altered the merging process, resulting in incorrect outputs. The test cases failed due to this mutation. Mutant 20 was effectively killed.

Effectiveness of Metamorphic Relations (MRs)

- **MR1 (Sorted List Relation)**:

    - **Effectiveness**: MR1 was effective in detecting Mutants 2, 6, and 13. These mutants introduced faults that disrupted the sorted output, making MR1 effective in identifying errors in the merging or appending processes. MR1 was not effective in detecting faults caused by Mutants 4, 8, 19, and 20, as these mutations did not directly affect the consistency of the sorted result under this relation.

- **MR2 (Invariant Under Permutations)**:

    - **Effectiveness**: MR2 effectively detected faults in Mutants 2, 4, 6, 8, 13, 19, and 20. These mutants failed to maintain the correct sorted order when the input list was permuted, demonstrating MR2’s strength in identifying issues related to permutation invariance and list ordering. MR2's effectiveness highlights its utility in catching errors where the handling of the list's order and consistency across different permutations is crucial.