Name: Phan Vu – Student ID: 104222099

Project Report

Due date: 2:59am, Monday, 5 Aug 2024

Task 1: Random Testing

Subtask 1.1: Understanding Random Testing Methodology

Intuition of Random Testing: Random testing involves the process of testing software by supplying randomly generated inputs to check for failures or errors. Its primary advantage is its simplicity and the breadth of coverage it can provide, potentially exposing edge cases that structured testing may miss.

Distribution Profiles for Random Testing

- **Uniform Distribution**: Every input within the specified range has an equal probability of being selected. This is often used when no prior knowledge about the input space is available.
- **Biased Distribution**: Some inputs are more likely to be selected than others based on certain criteria, such as user behavior patterns or areas known for previous errors.
- **Targeted Distribution**: Focuses on specific parts of the input space that are deemed critical or have higher chances of failure. This could be based on historical data or expert domain knowledge.

Process of Random Testing

1. **Define Input Space**: Clearly delineate the range and type of inputs that are valid for the software under test.
2. **Generate Inputs**: Use a random number generator to produce inputs according to the chosen distribution profile.
3. **Execute Test Cases**: Run the software with the randomly generated inputs and observe the outputs.
4. **Evaluate the Results**: Check outputs for correctness, performance issues, or failures.
5. **Refine and Repeat**: Based on the outcomes, refine the input generation strategy or distribution profile and repeat the testing to cover more scenarios.

Applications of Random Testing

- **Security Testing**: Random testing can be used to identify security vulnerabilities by randomly generating inputs that might include unexpected or malicious data.
- **Performance Testing**: Evaluating how software performs under stress conditions with random high-load inputs.
- **Usability Testing**: Seeing how systems handle unexpected user inputs randomly generated based on user interaction patterns.

Example Illustration

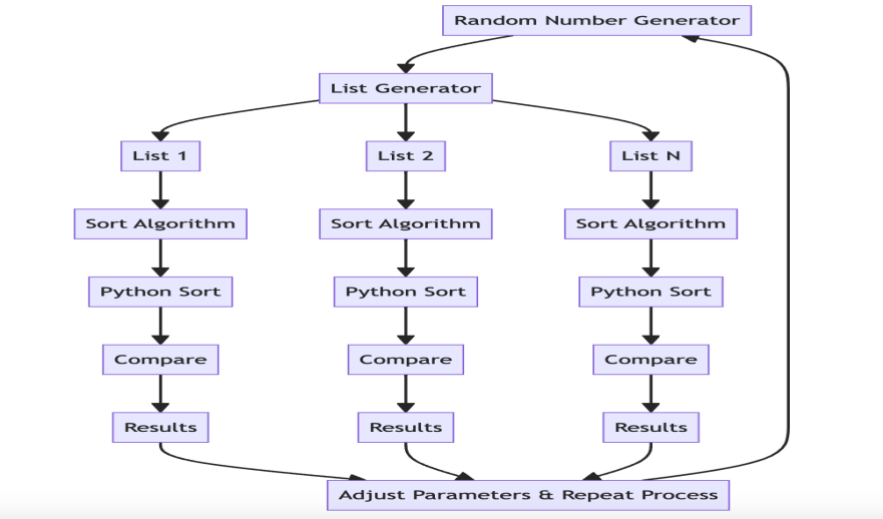

Figure 1: Diagram demonstrates how example work

Explanation of Each Step:

1. **Random Number Generator**: Initiates creation of random lists, defining size and range of integers.
2. **Generate Random Lists**: Outputs lists filled with random numbers.
3. **Custom Sorting Algorithm**: Each list is sorted using your custom algorithm.
4. **Python's sorted() Function**: Lists are also sorted using Python's built-in function for reference.
5. **Comparison**: Compares outputs from your sort and Python's sort for discrepancies.
6. **Results**: Indicates pass or fail based on match between custom-sorted and reference lists.
7. **Adjust Parameters**: Modifies list size or number range to refine the test based on results.
8. **Repeat**: Continues testing loop, possibly with adjusted parameters for better coverage.

Subtask 1.2: Application of Random Testing Methodology

Test Case Generation: Here, we will generate random lists of integers to use as test cases for a sorting program. Let's illustrate with two specific examples.

```
import random

# Function to generate a random list of integers
def generate_random_list(size, range_min, range_max):
    return [random.randint(range_min, range_max) for _ in range(size)]

# Example Test Case 1
test_case1 = generate_random_list(size=10, range_min=-100, range_max=100)
print("Test Case 1:", test_case1)

# Example Test Case 2
test_case2 = generate_random_list(size=20, range_min=-500, range_max=500)
print("Test Case 2:", test_case2)
```

Firgure 2: Program demonsrates two specific examples.

Explanation of the Test Cases

- **Test Case 1**: Generates a list of 10 random integers between -100 and 100. This tests how the sorting algorithm handles a moderate number of elements.
- **Test Case 2**: Generates a list of 20 random integers between -500 and 500, checking the algorithm's consistency with a larger list and wider range.

These test cases can then be used to verify that the sorting program correctly sorts the lists in ascending order. Each test would involve running the sorting program with the test case input and verifying the output against a known correct sorting method, such as Python's built-in sorted function.

```
# Asserting the sort function works as expected (assuming 'sort_function' is the function to test)
assert sort_function(test_case1) == sorted(test_case1), "Test Case 1 failed!"
assert sort_function(test_case2) == sorted(test_case2), "Test Case 2 failed!"
```

Figure 3: Program demonsrates two specific examples.

Task 2: Metamorphic Testing

Subtask 2.1: Understanding Metamorphic Testing Methodology

Metamorphic testing is an innovative approach designed to handle situations where determining the expected outcome (oracle) of a test case is challenging. It is particularly useful in systems where outputs are difficult to predict based on input alone, making traditional testing methods less effective.

Key Concepts in Metamorphic Testing:

1. **Test Oracle**: A test oracle is a mechanism used to determine the correctness of a program's output. In complex software systems, defining an oracle can be challenging or even impossible. Metamorphic testing addresses this issue by relying on relationships between outputs from multiple executions, rather than requiring an exact oracle.
2. **Untestable Systems**: Systems are considered untestable when it's impractical or impossible to determine the expected output for a given input under normal testing conditions. This is common with complex algorithms in machine learning or scientific computations where results are not deterministic or easily predictable.
3. **Motivation and Intuition**: The primary motivation behind metamorphic testing is to overcome the oracle problem in software testing. By identifying properties or relations (metamorphic relations) that the output should satisfy after undergoing certain modifications to the input, testers can ascertain the correctness of the program indirectly.
4. **Metamorphic Relations**:
    - These are relationships that are expected to hold between the outputs of two or more executions of the program under different sets of inputs. The inputs for these executions are related in a systematic way, and so are the outputs.
    - Example: For a sorting algorithm, a metamorphic relation might be that reversing the output list should yield the input list if it was sorted in descending order.
5. **Process of Metamorphic Testing**:
    - **Identify Metamorphic Relations**: Determine properties relating multiple inputs and their outputs.
    - **Generate Source Test Cases**: Create initial test cases as you normally would.
    - **Derive Follow-Up Test Cases**: Based on the metamorphic relations, modify the source test cases to create new, related test cases.
    - **Execute Tests**: Run both the source and follow-up test cases.
    - **Verify Relations**: Check if the expected metamorphic relations hold. If not, a defect might be present.
6. **Applications**:

- **Machine Learning Algorithms**: Used to verify the correctness of predictive models by altering input data and checking consistency in predictions.
- **Scientific Computing**: Useful for validating simulations or algorithms where outputs are complex and a traditional oracle is unavailable.
- **API Testing**: Can verify the behavior of APIs under varied conditions without requiring a fixed output for each input.

Example Illustration:

In these two below images, there are many things different of the same place, for example: time of photo taken is day vs night, there are different noise in the image such as some people vs many people, neon light vs natural light, no decoration vs a lot of decoration, … and the system is still expected to categorize the object correctly.


Figure 4: Photos taken of a violinist in 30/4 Park.

Here are some variances can be added to test this type of task:

Category	**Variance**	**Details**
Weather	Sunny	High brightness, shadow,…
	Rainy	Reduce visibility
	Thunderstorm	Adding noise
	CLoudy	Reduced visibility or adding noise
People	Pedestrian	Adding noise
	Motorcyclist	
	Rider	
	Hawker	
	…	
Decoration	Tree	Adding noise
	Flower	
	Banner	
	Flags	
	…	
Other	Light	Day, night, sunrise, sunset, artificial light, direct light,…
	Rotation	Straight face photo, slightly straight,…
	Scale	Big, small, ratio,…
	Nearby objects	Object next to, behind, on the left, on the right such as buildings, house,…

Combination Example		Old man play violin in the park in front of big building, portrait photo taken at sunny morning

Figure 4: Table demonstrate variances.

And here is how it will work:

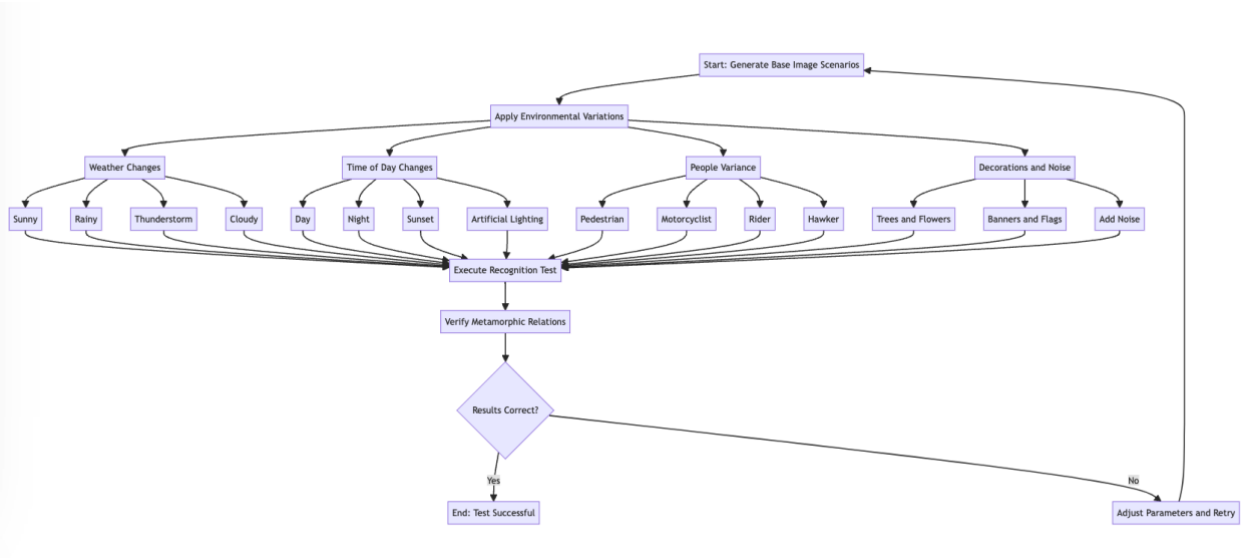

Figure 5: Digram show how example work.

Subtask 2.2: Metamorphic Relations for Testing a Sorting Program

We will define two metamorphic relations (MRs) for testing a sorting algorithm that handles non-empty lists of integers, including duplicates. These MRs will ensure the program's reliability and correctness without requiring predefined outputs, addressing the challenge of testing without a conventional oracle.

Metamorphic Relation 1: Reversal Invariance

Description: The sorted order of a list should be completely reversible. If you reverse a sorted list, and then sort it again, you should be able to reverse it back to its original sorted state.

Concrete Metamorphic Group:

- **Source Test Case**: Input a list of integers, e.g., [3, 1, 4, 1, 5].
- **Follow-Up Test Case**: Reverse the sorted output, then sort this reversed list.

Example:

- **Source List**: [3, 1, 4, 1, 5]
- **Sorted Source List**: [1, 1, 3, 4, 5]
- **Reversed Sorted List**: [5, 4, 3, 1, 1]
- **Resorted Reversed List**: [1, 1, 3, 4, 5]

Intuition: This MR tests the stability and consistency of the sorting algorithm. By ensuring that sorting a reversed sorted list yields the original sorted list, we validate that the sorting process is logically sound and handles order inversions correctly.

Metamorphic Relation 2: Idempotence of Sorting

Description: Sorting a list that has already been sorted should not change its order. This tests the idempotence of the sorting function, a fundamental property for sorting algorithms.

Concrete Metamorphic Group:

- **Source Test Case**: Input a randomly ordered list.
- **Follow-Up Test Case**: Sort the list, then sort the sorted list again.

Example:

- **Initial List**: [2, 3, 2, 8, 5, 1]
- **Sorted List**: [1, 2, 2, 3, 5, 8]
- **Resorted Sorted List**: [1, 2, 2, 3, 5, 8]

Intuition: This MR ensures that once a list is sorted, additional sorting operations should not alter its order. This relation is critical for verifying the algorithm's effectiveness in maintaining an already ordered list, thus confirming that the algorithm handles ordered inputs as expected without introducing any discrepancies.

Subtask 2.3: Comparing Random Testing and Metamorphic Testing

Both random testing and metamorphic testing are powerful techniques in the software testing arsenal, particularly useful when traditional testing approaches face limitations. Here's a comparative analysis highlighting the advantages and disadvantages of each, with a focus on aspects related to test oracles and test cases.

Attributes	Random Testing	Metamorphic Testing
Advantages	1. **Simple Implementation**: Minimal setup and planning required. 2. **Broad Coverage**: Explores a wide range of input scenarios, including rare edge cases. 3. **Scalability**: Can be automated and scaled up to handle large datasets and complex systems for stress testing.	1. **Oracle Problem Solution**: Useful where predicting the correct output is difficult. 2. **Structured Testing Approach**: Provides systematic and repeatable methods. 3. **Effective in Complex** Domains: Ideal for software in machine learning and data analytics.
Disadvantages	1. **Dependence on Oracle**: Needs a precise oracle to determine output correctness. 2. **Efficiency Concerns**: May generate redundant or irrelevant tests. 3. **Lack of Depth**: Might not target specific functionalities deeply.	1. **Reliance on Metamorphic Relations**: Effectiveness depends on appropriate MRs. 2. **Complex Setup**: Requires deep understanding to determine suitable MRs. 3. **Resource Intensity**: Implementing and maintaining tests based on MRs can be resource-intensive.
Oracle Requirements	**High**: Requires a known correct outcome for each input, challenging in complex systems.	**Low**: Focuses on output changes in response to input changes based on MRs, rather than exact outputs.
Test Case Characteristics	Independent and isolated; uses randomness to cover extensive input spaces without specific patterns.	Interdependent; follow-up test cases derive from initial test results, emphasizing relationships established by MRs.

Figure 6: Comparision Between Random Testing and Metamorphic Testing.

In conclusion, while both testing methodologies offer unique benefits, the choice between random testing and metamorphic testing should be based on the specific requirements of the software being tested, including the availability of an oracle, the nature of the software's functionality, and the depth of testing required.

Task 3: Test A Program Of Your Choice.

Select and Describe the Program

The program in focus is a Python implementation of the **counting_sort** function. This function sorts a list of integers, which can include positive, negative, or zero values.

Source: https://github.com/TheAlgorithms/Python/blob/master/sorts/counting_sort.py

File name: **counting_sort.py**

Define Metamorphic Relations

1. **Element Duplication (Metamorphic Relation)**

   Metamorphic relations are properties of the target function that remain consistent under certain transformations of the input. These relations help in generating additional test cases and validating the correctness of the function. The metamorphic relations defined for the **counting_sort** function are:

   - **Original Input:** A list [1, 2, 3]
   - **Transformed Input:** A duplicated list [1, 2, 3, 1, 2, 3]
   - **Expected Result:** The sorted output should also show duplication appropriately: [1, 1, 2, 2, 3, 3].

2. **Scale Elements (Metamorphic Relation)**

   - **Original Input:** A list [3, 1, 4, 1, 5]
   - **Transformed Input:** Each element is scaled (multiplied) by 2, resulting in [6, 2, 8, 2, 10].
   - **Expected Result:** The sorted result of the scaled list should match the scaled version of the original sorted list.

3. **Concatenation of Sorted Arrays (Metamorphic Relation)**

   - **Original Input:** Two separate sorted lists [1, 2, 3] and [4, 5, 6]
   - **Transformed Input:** A concatenated list [1, 2, 3, 4, 5, 6]
   - **Expected Result:** Sorting the concatenated list should naturally yield [1, 2, 3, 4, 5, 6], as both sublists are already sorted and just concatenated.

Implement Metamorphic Testing

The following metamorphic tests are implemented to validate the **counting_sort** function:

```python
import unittest
from counting_sort import counting_sort

class TestCountingSort(unittest.TestCase):

    def test_basic_functionality(self):
        self.assertEqual(counting_sort([4, 2, 5, 2, 3]), [2, 2, 3, 4, 5])

    def test_empty_list(self):
        self.assertEqual(counting_sort([]), [])

    def test_negative_numbers(self):
        self.assertEqual(counting_sort([-2, -5, -45]), [-45, -5, -2])

    def test_all_same_numbers(self):
        self.assertEqual(counting_sort([5, 5, 5, 5]), [5, 5, 5, 5])

    def test_element_duplication(self):
        original = [1, 2, 3]
        duplicated = original * 2  # Duplicate elements
        self.assertEqual(counting_sort(duplicated), [1, 1, 2, 2, 3, 3])

    def test_scale_elements(self):
        original = [3, 1, 4, 1, 5]
        scaled = [x * 2 for x in original]
        expected = [x * 2 for x in counting_sort(original)]
        self.assertEqual(counting_sort(scaled), expected)

    def test_concatenation_sorted_arrays(self):
        sorted_a = [1, 2, 3]
        sorted_b = [4, 5, 6]
        concatenated_and_sorted = counting_sort(sorted_a + sorted_b)
        self.assertEqual(concatenated_and_sorted, sorted_a + sorted_b)


if __name__ == "__main__":
    unittest.main()

```

Figure 7: Metamorphic tests are implemented

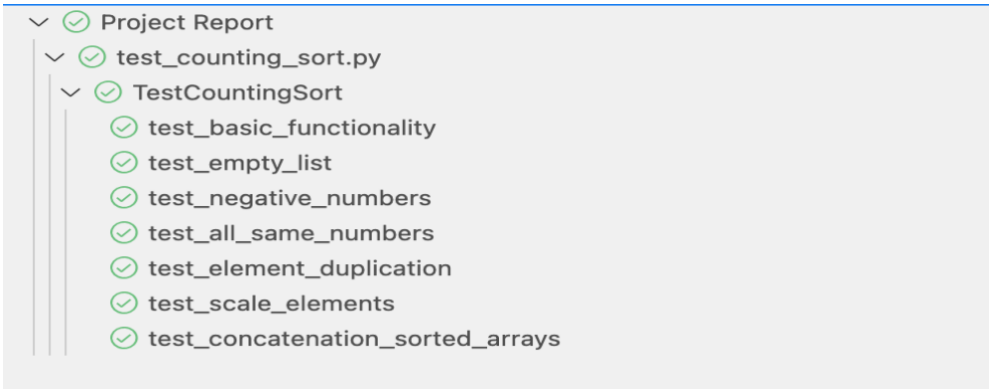

Figure 8: The original test result

Generate Mutants

The tool MutPy was used to generate mutants. MutPy is a mutation testing framework for Python, which automatically creates mutants by applying mutation operators to the code. The command used to generate mutants is:

mut.py --target counting_sort --unit-test test_counting_sort -m

Execute Metamorphic Tests on Mutants

The effectiveness of metamorphic testing is highlighted by the ability of MRs to detect faults in mutants. A summary table showing the mutation types, the code changes, and the test outcomes would effectively illustrate which mutations were detected (killed) and which were not (survived).

Mutant ID	Mutation Type	Original Code Snippet	Mutated Code Snippet	Status	Result	Execution Time (s)
1	AOR	coll_max + 1 - coll_min	coll_max - 1 - coll_min	Killed	6 failed, 1 passed	0.18976
2	AOR	coll_max + 1 - coll_min	coll_max + 1 + coll_min	Killed	1 failed, 6 passed	0.18264

3	AOR	[0] * counting_arr_length	[0] / counting_arr_length	Killed	6 failed, 1 passed	0.18391
4	AOR	[0] * counting_arr_length	[0] // counting_arr_length	Killed	6 failed, 1 passed	0.18365
5	AOR	[0] * counting_arr_length	[0] ** counting_arr_length	Killed	6 failed, 1 passed	0.18675
6	AOR	counting_arr[number - coll_min] -= 1	counting_arr[number + coll_min] -= 1	Survived	7 passed	0.18174
7	AOR	counting_arr[number - coll_min] += 1	counting_arr[number + coll_min] += 1	Killed	6 failed, 1 passed	0.18394
8	AOR	counting_arr[i] + counting_arr[i - 1]	counting_arr[i] - counting_arr[i - 1]	Killed	5 failed, 2 passed	0.18417
9	AOR	counting_arr[i] + counting_arr[i - 1]	counting_arr[i] + counting_arr[i + 1]	Killed	5 failed, 2 passed	0.18355
10	AOR	[0] * coll_len	[0] / coll_len	Killed	6 failed, 1 passed	0.18299
11	AOR	[0] * coll_len	[0] // coll_len	Killed	6 failed, 1 passed	0.18286
12	AOR	[0] * coll_len	[0] ** coll_len	Killed	6 failed, 1 passed	0.18362
13	AOR	ordered[counting_arr[collection[i] - coll_min] - 1] = collection[i]	ordered[counting_arr[collection[i] + coll_min] - 1] = collection[i]	Killed	6 failed, 1 passed	0.18432
14	AOR	ordered[counting_arr[collection[i] - coll_min] - 1] = collection[i]	ordered[counting_arr[collection[i] - coll_min] + 1] = collection[i]	Killed	6 failed, 1 passed	0.18204
15	AOR	counting_arr[collection[i] - coll_min] -= 1	counting_arr[collection[i] + coll_min] -= 1	Killed	6 failed, 1 passed	0.18469
16	ASR	counting_arr[collection[i] - coll_min] -= 1	counting_arr[collection[i] - coll_min] += 1	Killed	3 failed, 4 passed	0.18368
17	ASR	counting_arr[collection[i] - coll_min] += 1	counting_arr[collection[i] - coll_min] -= 1	Survived	7 passed	0.18297
18	COI	if collection == []	if not (collection == [])	Killed	3 failed, 4 passed	0.18415
19	COI	if number == 7	if not (number == 7)	Killed	6 failed, 1 passed	0.18537
20	COI	if **name** == '**main**'	if not (**name** == '**main**')	Survived	7 passed	0.18468
21	ROR	if collection == []	if collection != []	Survived	7 passed	0.18537
22	ROR	if number == 7	if number != 7	Killed	6 failed, 1 passed	0.18558
23	ROR	if **name** == '**main**'	if **name** != '**main**'	Killed	6 failed, 1 passed	0.18582
24	ROR	assert counting_sort_string('thisisthestring') == 'eghhiiinrsssttt'	assert counting_sort_string('thisisthestring') != 'eghhiiinrsssttt'	Survived	7 passed	0.18537

Figure 9: The results from executing these tests on various mutants.

```
vufanity@Vus-Mac-Buc-Po-Ro Project Report % /usr/local/bin/python3 "/Users/vufanity/SWE30009/Project Repo
rt/mutant_automatic.py"
Mutant #1 killed
Mutant #2 killed
Mutant #3 killed
Mutant #4 killed
Mutant #5 killed
Mutant #6 survived
Mutant #7 killed
Mutant #8 killed
Mutant #9 killed
Mutant #10 killed
Mutant #11 killed
Mutant #12 killed
Mutant #13 killed
Mutant #14 killed
Mutant #15 killed
Mutant #16 killed
Mutant #17 survived
Mutant #18 killed
Mutant #19 killed
Mutant #20 survived
Mutant #21 survived
Mutant #22 killed
Mutant #23 killed
Mutant #24 survived
Detailed test results are saved in: mutants/results
```

Figure 10: The result from terminal after being executed.

Conclusion

The metamorphic testing process demonstrated its utility in identifying faults that are not easily detectable through conventional testing approaches. The mutations introduced by MutPy provided a robust platform for evaluating the resilience of the **counting_sort** function against logical and arithmetic errors. The results underscore the importance of using systematic transformations of inputs to uncover hidden bugs in the implementation.

Test Coverage and Quality:

- The high percentage of killed mutants (75%) reflects good test coverage and quality of the metamorphic tests.
- The survived mutants highlight areas for improvement in the test cases, particularly in covering edge cases and specific conditions that might not have been fully tested.

Recommendations

For further enhancement of the testing strategy:

- Expand the set of metamorphic relations to cover more complex transformations.
- Increase the diversity of input data to include more edge cases and unusual combinations.
- Consider integrating automated tools for continuous testing and integration, facilitating regular feedback on code changes.