Name: Trac Duc Anh Luong - ID: 103488117

Project Report

Task 1

Subtask 1.1 Random testing methodology

1. The intuition of random testing
    a. Random testing is a black-box approach in which the test cases are randomly and independently selected from the entire input domain randomly and independently.
    b. These inputs do not utilise any information to guide the selection of test cases.
2. Distribution profiles for random testing
    a. Uniform distributions: each input is of equal selection probability
    b. Operational distributions (profiles): selection probability follows the probability of being used.
    c. Example: If we have a program that processes the input age of a user, implementing uniform distributions will randomly select the age from 0 to 100 with equal probability. Using operational distribution, we could assign a higher probability to the most common age group (e.g., 25-64). In contrast, less common age groups (e.g., 65+) will be assigned a lower probability, simulating real-world scenarios more closely.
3. The process of random testing
    a. Identify the input domain & program constraints: Understand the constraints and range of valid inputs for the program under test.
    b. Implement an RG (Random Generator) to select random test inputs: Inputs will be independently chosen from the defined input domain.
    c. Test execution: Run the program under test with the selected random inputs.
    d. Result comparison: Compare the actual and expected output based on the program's requirements.
    e. Handle defects: If any test case fails, we must investigate where the defect occurred and take necessary corrective actions.
4. Applications
    a. Software Development: Random testing can exploit edge cases of programs during the testing phase.
    b. Security Testing: Security professionals can utilise random testing to find vulnerabilities in a program by inputting it with random and malformed data.
    c. Quality Assurance: By using unexpected inputs, we can find the program's weaknesses and improve its robustness before release to the end-user.
    d. Complementing other testing methods: Random testing can be used together with other testing methods to broaden the testing coverage and ensure the program is not faulty.

Subtask 1.2 Apply random testing

1. Identify the input domain & program constraints:
    a. The program takes in a list of integers

→ Elements in the list cannot be decimals.
→ Elements in the list can be negative, positive, or 0.
→ Elements in the list can be either odd or even.

b. The list must not be empty.
c. The list can contain duplicate numbers.

2. Implement an RG (Random Generator) to select random test inputs

Code:
```
import random

def generate_test_cases(num_cases, min_length, max_length, min_value, max_value):
  test_cases = []
  for _ in range(num_cases):
    length = random.randint(min_length, max_length)
    test_case = [random.randint(min_value, max_value) for _ in range(length)]
    test_cases.append(test_case)
  return test_cases

test_cases = generate_test_cases(num_cases=2, min_length=1, max_length=10,
min_value=-100, max_value=100)
for case in test_cases:
  print(case)
```

Output:
```
[70, 35, -4, 58, -6, 84, -19, 76, 61, 55, 35]
[42, -54, -61, 47, 0, 100, -72, -61, -81]
```

Test case 1:
- Input: [70, 35, -4, 58, -6, 84, -19, 76, 61, 55, 35]
- Expected Output: [-19, -6, -4, 35, 35, 55, 58, 61, 70, 76, 84]

Test case 2:
- Input: [42, -54, -61, 47, 0, 100, -72, -61, -81]
- Expected Output: [-81, -72, -61, -61, -54, 0, 42, 47, 100]

From the 2 test cases above, we can see that the input covers a wide range of requirements, including:
- Odd numbers
- Even numbers
- Positive numbers
- Negative numbers
- Duplicates

These created test cases show how various and objective test inputs for a sorting algorithm may be produced using the random testing methodology. Random testing assists in guaranteeing the robustness and accuracy of the sorting algorithm by including an extensive array of potential inputs, including duplicates and edge cases.

Task 2

Subtask 2.1. Metamorphic testing methodology

Metamorphic testing (MT) is a property-based software testing methodology that addresses the problem of unreliable test oracles and test case generation. This method is often used for testing untestable systems, where the system complexity makes it impossible or highly resource-costly to determine the correctness of each output.

1. Test oracle: The test oracle is a set of rules and mechanisms for determining the correct output of a given input and whether a test has passed or failed. In conventional software testing, test oracles are often derived from system specifications or expected behaviours.

   Example: For a program that accepts 2 number inputs and returns the sum, a test oracle can be:
   Input: 3 + 5
   Expected Output (Oracle): 8

2. Untestable systems: when it comes to "non-testable" or "oracle problem" systems, creating an accurate and reliable test oracle is impractical or impossible. This can occur due to numerous reasons:
   a. Complex computations: The system performs calculations that are too complex to verify manually (test oracles are available but too expensive to be applied).
   b. Lack of specifications: There needs to be a clear definition of expected behaviour (absence of test oracle).
   c. Subjective outputs: The system produces results that require human judgement.

   Examples: highly complex systems, such as simulations, machine learning algorithms, embedded systems, natural language processing (NLP) systems

3. The motivation and intuition of metamorphic testing
   a. Intuition: Metamorphic testing was created to solve the oracle problem in untestable systems. The essential insight is that, even though we might not be aware of the precise right result for a certain input, we typically understand how changes to the input should affect the output.
   b. Motivation:
      i. To test systems where traditional test oracles are unavailable or impractical.
      ii. To increase test coverage and find subtle bugs that conventional testing methods might miss.
      iii. To automate the testing process for complex systems.

4. Metamorphic relations
   A metamorphic relation (MR) is a property of the system under test or algorithm to be implemented that describes how the output should change when the input is transformed in a specific way. It is a relationship between multiple inputs and their corresponding outputs.

Example: Consider a program that accepts a list of integers and returns their sum.
If we add a constant element to the list, the follow-up output should be the sum of the source output + the constant element.
Source Input: [1, 2, 3, 4]
Source Output: 10
Expected follow-up Input: [1, 2, 3, 4, 10]
Expected follow-up Output: 10 + 10 = 20

5. The process of metamorphic testing
    a. Utilising certain test case selection techniques, define and run source (initial) test cases.
    b. List important properties of the program (also known as the MRs).
    c. Create and run additional test cases based on the original test cases, referencing the pre-defined metamorphic relations.
    d. Using the calculated results, confirm the metamorphic relations.

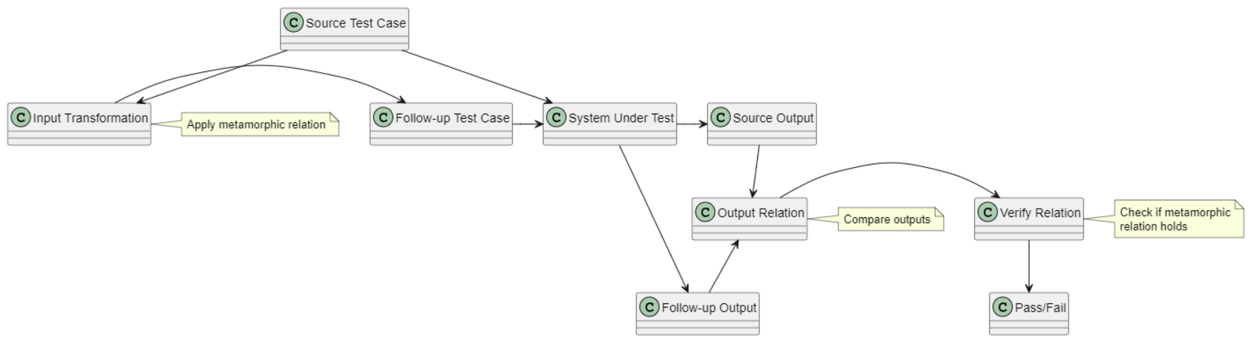


Figure 1: The process of metamorphic testing

6. Applications
    a. Machine Learning and AI: Testing classification algorithms, neural networks, and other ML models.
    b. Numerical Programs: Testing mathematical software and scientific simulations.
    c. Compilers: Testing compiler optimisations and code generation.
    d. Graphics Rendering: Testing computer graphics algorithms.

Example of MT application: For an image classification system, rotating an image shouldn't change its classification (rotation invariance).

Subtask 2.2. Apply metamorphic testing to the same program as subtask 1.2

MR1: Unchanged Permutation

1. **Description**: If we permute the elements of the source input list, the follow-up output list and source output list should remain the same.
2. **Intuition**: The final sorted output shouldn't be impacted by the elements' order in the input list. An algorithm for sorting numbers correctly should always result in the same sorted list, no matter how the numbers are initially organised.
3. **How it works**:

   a. Take the original input list as the source test case.
   b. Create a follow-up test case by randomly permuting the elements of the source list.
   c. Apply the sorting algorithm to both the source and follow-up test cases.
   d. Verify that the outputs from both cases are identical.
4. **Concrete Metamorphic Group**:
   - Source test case: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
   - Follow-up test case (permutation): [5, 3, 5, 2, 1, 1, 9, 4, 6, 3]
   - Expected output for both: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
   - The MR will be satisfied when the program returns the same expected output for both the SI and FI lists.

MR2: Unchanged Additive Shift

1. **Description**: If we add a constant value to all elements in the source input list, the follow-up output should be the same list as the source output list with the constant added to each element, maintaining the same order.
2. **Intuition**: The relative order of all integers remains unchanged when a constant is added. As a result, each element in the sorted output should increase by the constant while keeping the same order.
3. **How it works**:
   a. Take the original input list as the source test case.
   b. Create a follow-up test case by adding a constant value to each element in the source list.
   c. Apply the sorting algorithm to both the source and follow-up test cases.
   d. Verify that the constant plus the matching element from the source output equals each element in the follow-up output list.
4. **Concrete Metamorphic Group**:
   - Source test case: [1, 4, 3, 9, 5, 8, 6, 2, 8, 0]
   - Constant to add to each element: 10
   - Follow-up test case: [11, 14, 13, 19, 15, 18, 16, 12, 18, 10]
   - Source output: [0, 1, 2, 3, 4, 5, 6, 8, 8, 9]
   - Follow-up output: [10, 11, 12, 13, 14, 15, 16, 18, 18, 19]
   - The MR will be satisfied when the sorting program produces outputs that satisfy this relation (each element in the FO is 10 more than the corresponding element in the SO).

Subtask 2.3. Advantages and disadvantages of random testing and metamorphic testing comparison

Aspect	**Random Testing**	**Metamorphic testing**
Test Oracle		
Advantages	- Simpler test oracle - Easy to implement - Uses straightforward, predefined expected outputs	- Handles the oracle problem - Flexible (uses relations between inputs and outputs) - Can test systems with unknown or hard-to-verify outputs
Disadvantages	- Requires a complete test oracle	- More complex oracle

	- Limited applicability for complex systems - Not suitable when correct output is difficult to determine	- Requires understanding and defining metamorphic relations - Potential for false positives if relations are incorrectly defined
Test Cases		
Advantages	- Easy generation - Unbiased coverage - Can explore unexpected areas of input space	- Systematic coverage - Efficient (each case serves a specific purpose) - Derived based on system properties
Disadvantages	- May generate redundant or irrelevant cases - Poor coverage guarantee - No assurance of covering all important scenarios	- More complex generation - Requires careful design of follow-up cases - Potential bias based on defined relations
Implementation Complexity		
Advantages	- Straightforward to implement - Minimal setup required - Can use existing random generators	- Reusable metamorphic relations - Can be automated once relations are defined
Disadvantages	- May require large number of test runs - Difficulty in reproducing specific test cases	- Requires expertise to define good metamorphic relations - More complex test harness needed
Fault Detection		
Advantages	- Can detect unexpected faults - Good for finding crashes and obvious errors	- Effective at finding subtle logical errors - Can detect faults in complex computations
Disadvantages	- May miss systematic errors - Less effective for logical or algorithmic faults	- May miss simple crashes if not covered by relations - Effectiveness depends on quality of defined relations
Applicability		
Advantages	- Universally applicable - Useful for most types of software	- Particularly useful for numerical programs, AI/ML systems - Effective for testing properties rather than specific outputs
Disadvantages	- Less effective for programs with complex logic - Not ideal for testing specific properties	- Requires system to have identifiable metamorphic properties - Less useful for simple, deterministic systems

Test Result Analysis		
Advantages	- Straightforward pass/fail criteria - Easy to automate result checking	- Can provide insights into system properties - Helps in understanding system behaviour
Disadvantages	- May produce large volumes of results to analyse (redundancy) - Can be hard to identify patterns in failures	- Requires careful interpretation of relation violations - May need domain expertise to analyse results
Overall		
Advantages	- Simple to implement and automate - Can find unexpected bugs - Useful for stress and performance testing	- Effective for complex systems and oracle problems - Provides targeted and meaningful test cases - Tests properties and relationships
Disadvantages	- May miss critical bugs - Less effective for complex systems - Requires complete test oracle	- Requires upfront analysis for relations - May miss bugs unrelated to defined relations - More time-consuming initial setup
Best Suited For	- Systems with manageable input spaces - When complete test oracle is available - Stress testing and finding unexpected edge cases	- Complex systems with oracle problems - Scientific computing, machine learning, and embedded systems - Testing specific properties and behaviours

Figure 2: Random testing and metamorphic testing comparison

Task 3

Program: Binary Search Algorithm
Source: https://github.com/TheAlgorithms/Python/blob/master/searches/binary_search.py
Language: Python

1. **Proposed metamorphic relations**

   MR1: Monotonicity

   a. Definition: If a value x is less than another value y in the input array, then the index returned by the binary search for x should be less than the index returned for y.
   b. Intuition: This relation captures the instrumental property of a sorted array: elements are ordered sequentially.
   c. Example: For a sorted array [1, 2, 5, 7, 8, 10], if x = 2 and y = 7, then index(2) < index(7)

   MR2: Subarray search

   a. Definition: If a value x is found within a subarray of the original sorted array, the binary search on the subarray should return the same relative index within the subarray.
   b. Intuition: The ordering property of the array is preserved in its subarrays.

c. Example: For a sorted array [1, 2, 5, 7, 8, 10], if target = 7 is found at index = 3, then for the subarray [1, 2, 5, 7], target = 7 should still be found at index 3.

2. **Implementation of metamorphic testing**
3. **Mutants generation**
   Here are 20 mutants for the provided binary search function:

No.	Original Program	Mutant
Arithmetic Operators		
1	`midpoint = left + (right - left) // 2`	`midpoint = left + (right + left) // 2`
2	`midpoint = left + (right - left) // 2`	`midpoint = left - (right - left) // 2`
3	`midpoint = left + (right - left) // 2`	`midpoint = left + (right / left) // 2`
Comparison Operators		
4	`left <= right`	`left < right`
5	`left <= right`	`left >= right`
6	`current_item == item`	`current_item != item`
7	`item < current_item`	`item <= current_item`
8	`item < current_item`	`item > current_item`
Assignment Operators		
9	`left = 0`	`left = 1`
10	`right = len(sorted_collection) - 1`	`right = len(sorted_collection)`
11	`right = midpoint - 1`	`right = midpoint`
12	`left = midpoint + 1`	`left = midpoint + 5`
Logical Operators		
13	`if list(sorted_collection) != sorted(sorted_collection):`	`if list(sorted_collection) == sorted(sorted_collection):`
14	`if current_item == item:`	`if current_item == item or current_item == item + 1:`
Return Statements		
15	`return midpoint`	`return midpoint + 1`
16	`return midpoint`	`return midpoint - 1`

17	`return -1`	`return 0`
Conditional Statements		
18	`if list(sorted_collection) != sorted(sorted_collection):`	`Remove the entire block`
19	`elif item < current_item:`	`Change the condition to else:`
20	`if current_item == item:`	`Add an else block after the condition and return a random value`

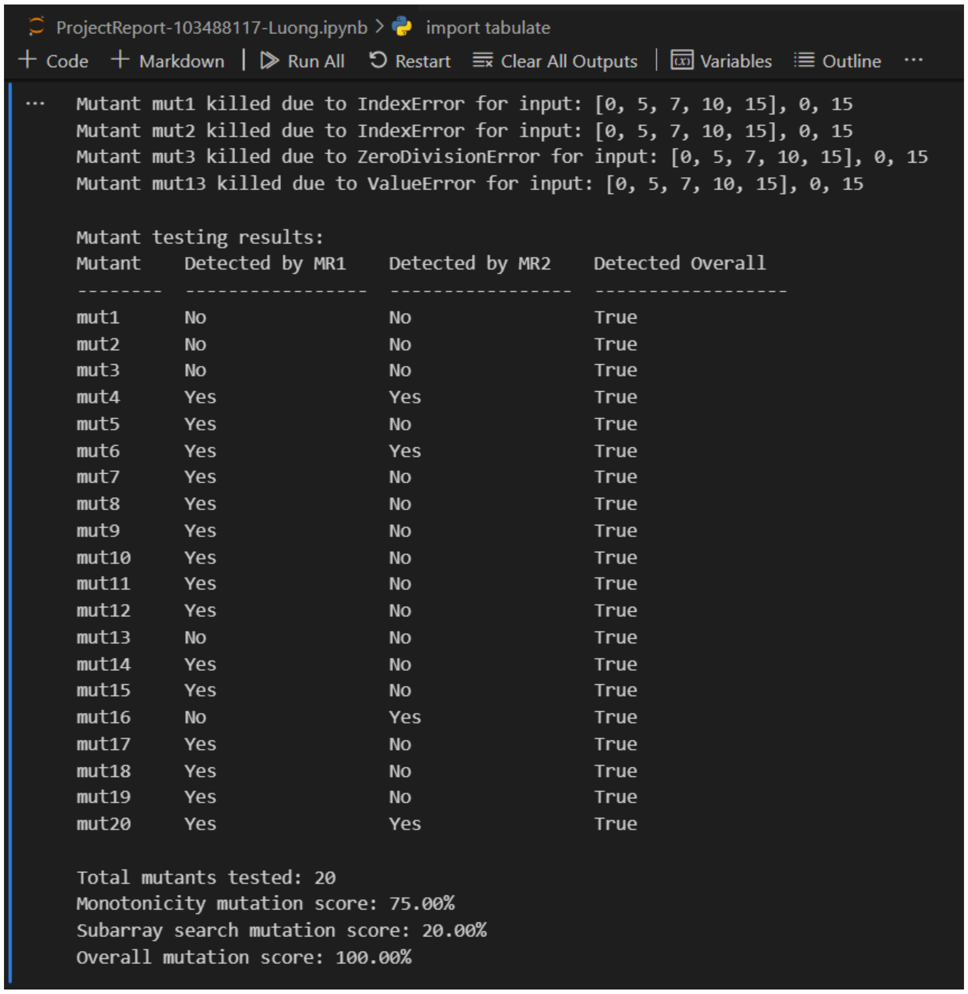
Figure 3: Mutants generation

4. **Testing process description**
    a. Mutation Testing:
        i. 20 mutants were created from the original binary search implementation.
        ii. Each mutant was tested against both MRs using various input scenarios.
        iii. A mutant was considered "killed" if it produced an error or violated an MR.
    b. Evaluation Metrics:
        i. Individual MR effectiveness
        ii. Overall mutation score
        iii. Types of errors detected

(next page)

5. **Results**

a. Screenshot of program being executed:


ProjectReport-103488117-Luong.ipynb > import tabulate
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...

```
Mutant mut1 killed due to IndexError for input: [0, 5, 7, 10, 15], 0, 15
Mutant mut2 killed due to IndexError for input: [0, 5, 7, 10, 15], 0, 15
Mutant mut3 killed due to ZeroDivisionError for input: [0, 5, 7, 10, 15], 0, 15
Mutant mut13 killed due to ValueError for input: [0, 5, 7, 10, 15], 0, 15

Mutant testing results:
Mutant    Detected by MR1    Detected by MR2    Detected Overall
--------  -----------------  -----------------  ------------------
mut1      No                 No                 True
mut2      No                 No                 True
mut3      No                 No                 True
mut4      Yes                Yes                True
mut5      Yes                No                 True
mut6      Yes                Yes                True
mut7      Yes                No                 True
mut8      Yes                No                 True
mut9      Yes                No                 True
mut10     Yes                No                 True
mut11     Yes                No                 True
mut12     Yes                No                 True
mut13     No                 No                 True
mut14     Yes                No                 True
mut15     Yes                No                 True
mut16     No                 Yes                True
mut17     Yes                No                 True
mut18     Yes                No                 True
mut19     Yes                No                 True
mut20     Yes                Yes                True

Total mutants tested: 20
Monotonicity mutation score: 75.00%
Subarray search mutation score: 20.00%
Overall mutation score: 100.00%
```

Figure 4: Program execution

b. Effectiveness of Metamorphic Relations:

MR	**Mutants Detected**	**Mutation Score**
Monotonicity	15	75%
Subarray	4	20%
Overall	20	100%

Figure 5: Mutation Detection by MR

Analysis:

- Monotonicity (MR1) was highly effective, detecting 75% of the mutants.
- Subarray search (MR2) was less effective, detecting only 20% of the mutants.
- Combined, the two MRs achieved a 100% mutation score, detecting all mutants.

c. Types of Errors Detected:

Error Types	**Count**	**Mutant**
IndexError	2	mut1, mut2
ZeroDivisionError	1	mut3
ValueError	1	mut13
Logical Errors	16	mut4-mut12, mut14-mut20

Figure 6: Error types

Analysis:
- Four mutants (mut1, mut2, mut3, mut13) were detected due to runtime errors.
- The majority of mutants (16) were detected due to logical errors that violated the MRs.

d. Complementary Nature of MRs:

Detection Scenario	**Count**	**Percentage**
Only by MR1	12	60%
Only by MR2	1	5%
By both MR1 and MR2	3	20%
By neither (errors)	4	15%

Figure 7: MR Complementarity

Analysis:
- MR1 (Monotonicity) was more effective overall, uniquely detecting 60% of mutants.
- MR2 (Subarray search) uniquely detected one mutant that MR1 missed.
- 15% of mutants were detected by both MRs, showing some overlap in effectiveness.
- 20% of mutants caused runtime errors, detected without needing MR verification.

6. Discussion

a. Effectiveness of MRs

MR1 (Monotonicity relation) proved highly effective and could detect a wide range of logical errors in the binary search implementation. This implies that checking the order preservation property is instrumental for validating binary search and other searching algorithms. MR2 (subarray search relation) was less effective but still contributed to the overall mutation score by detecting some of the mutants that MR1 missed. From this test, we can see the importance of using multiple, diverse MRs in testing.

b. Complementarity of MRs
   The two MRs demonstrated significant complementarity and achieved a 100% mutation score. This highlights the benefit of using several MRs to evaluate various algorithm behaviour characteristics.
c. Types of Errors Detected
   Some mutants were discovered due to runtime issues, while most were due to MR violations. This shows that metamorphic testing can help find implementation problems that cause runtime failures and logical mistakes.
d. Limitations and Future Work
   With different types of input data or more significant input sizes, the MRs' efficiency may change. More testing with a variety of datasets would be helpful.

7. Conclusion

The metamorphic testing approach, which uses the Monotonicity and Subarray search relations, proved very successful when testing the binary search implementation. The 100% mutation score shows how effective this method is at identifying runtime problems in addition to logical faults. The two MRs' complementary qualities emphasise how important it is to use a variety of relations when conducting metamorphic testing. To further confirm and enhance the resilience of binary search implementations, future work might concentrate on increasing the list of MRs and testing with more varied input scenarios.