Vu Duc Tran 104175614

Task 1:

1.1

Effectiveness metric can be defined as the measure of how effectively testing is executed. It is a necessary tool for evaluating how well testing technique can spot bugs and defects by quantifying the performance of testing strategies. Different metrics help identify different strengths and weaknesses of testing based on different factors.

Three commonly used metrics are:

i. **P-measure**:
   a) This metric calculates the probability of detecting at least one failure-causing input during the testing process.
   b) This metric is defined as the ratio of failure-causing inputs to the total number of inputs.

ii. **E-measure**:
   a) This metric defines the expected number of failures found during testing.
   b) This metric calculates the expected number of failures and gives an idea of how many of issues are expected to be encounter based on the number of test cases used.

iii. **F-measure**:
   a) This measure provides the expected number of test cases to detect the first failure.
   b) F-Measure combines both metrics into one score (F-Score) that shows the balance between precision and recall.
      - **Precision**: measuring the ratio of true positives to the total predicted positives. It addresses how many identified failures are actual failures.
      - **Recall**: considering the ratio of true positives to the total actual positives. It answers how many actual failures were detected.

From the given a program with an input domain consisting of 3 partitions of sizes 100, 200, and 250. Assume there are 20 test cases in total: 3, 5, and 2 failure-causing inputs, respectively in these partitions.

Hence, we have:

- $n = 20$
- $d_1 = 100$, $d_2 = 200$, $d_3 = 250$
- $m_1 = 3$, $m_2 = 5$, $m_3 = 2$

Total failure-causing inputs: $m = m_1 + m_2 + m_3 = 3 + 5 + 2 = 10$

Total size: $d = d_1 + d_2 + d_3 = 100 + 200 + 250 = 550$

Partition Case Distribution:

- $n_1 = (100/550) * 20 = 3.64 \approx 4$ (test cases)
- $n_2 = (200/550) * 20 = 7.27 \approx 7$ (test cases)
- $n_3 = (250/550) * 20 = 9.09 \approx 9$ (test cases)

I will illustrate the effectiveness of random testing and partition testing with P-measure and E-measure.

Random Testing	Partition Testing
P - measure	

$P_r = 1 - (1 - \frac{10}{550})^{20} \approx 0{,}3072$	$P_p = 1 - (1 - \frac{3}{100})^4 * (1 - \frac{5}{200})7 * (1 - \frac{2}{250})^4 \approx 0{,}3102$
E - measure	
$E_r = \frac{10}{550} * 20 \approx 0{,}3636$	$E_r = \frac{3}{100} * 4 + \frac{5}{200} * 7 + \frac{2}{250} * 9 \approx 0{,}367$

Table 1. Random vs. Partition Testing Measures

From the above table, we can see the proportional sampling strategy leads to a more balanced selection of test cases relative to the size and potential failure rates of each partition. Regarding to the outcomes, since the partition case distribution is approximate not exact, the results may not fully demonstrate the advantages of PSS (proportional sampling strategy):

- The P-measure of PSS is not less than that of random testing.
- The E-measures of PSS and random testing are approximately equal.

1.2

A test oracle is a mechanism or method to check whether the output of a program on a given input is correct or not. In traditional testing, oracles are used to validate results. However, there are untestable or non-testable systems that the outputs of some inputs cannot be verified. Hence, Metamorphic testing can effectively address these challenges by leveraging MRs to assess consistency across related inputs.

Metamorphic Testing is a testing technique particularly useful programs where traditional testing approaches may be insufficient, specifically in untestable programs or when there is no available test oracle assessing the correctness of outputs. But metamorphic can still be applicable even if test oracle exists.

The primary motivation for metamorphic testing lies in the need for reliable testing in environments where traditional methods fail. The intuition behind metamorphic testing is that although an individual input's outcome may not be known, the relationship between inputs and outputs can frequently be determined. For instance, you are having a list of numbers and want to the biggest number among them, you can expect the output does not change regardless of the order of the list.

Metamorphic Relations (MRs): are key to metamorphic testing; they are crucial necessary properties involving multiple related inputs and their outputs.

Process of Metamorphic Testing:

i. Define and execute source test cases: using some test case selection strategies.
ii. Identify metamorphic relations.
iii. Construct follow-up test cases
iv. Execute tests and verify outputs.

Applications of Metamorphic Testing:

i. Data encryption tool encrypts text messages
   a) Test cases

- Test case 1: plain text “hello” => Expected output: A_1
- Test case 2: plain text “bonjour” => Expected output: A_2

b) MRs
- MR1 - Concatenation of Two Inputs: If we encrypt "hello" and "bonjour" separately, concatenating their ciphertexts should equal the ciphertext of "hellobonjour".
- MR2 - Repeating the Input String: The length of the ciphertext for "hellohello" should be roughly double that of "hello".

c) Follow-up test cases
- Follow-Up for MR1: The ciphertext of "hellobonjour" should match the concatenated ciphertext of A_1 and A_2.
- Follow-Up for MR2: The length of ciphertext of "hellohello" should be double that of the concatenated ciphertext of A_1.

d) Verify outputs
- Test for MR1: output: A_3. Verify if A_3 is equal to the results of A_1 and A_2.
- Test for MR2: output: A_4. Verify if A_4 is two times longer than the results of A_1.

ii. Image classification model

a) Test cases
- Test case 1: a clear picture of a dog => Expected output: “dog”

b) MRs
- MR1 – Rotating the image: Rotating an image should not change the classification.
- MR2 – Scaling the image: Resizing the image to a smaller size should still lead to the same result.

c) Follow-up test cases
- Follow-Up for MR1: Rotating the picture by 90 degrees.
- Follow-Up for MR2: Scaling the picture down to half size.

d) Verify outputs
- Test for MR1: output: “dog”
- Test for MR2: output: “dog”

1.3

Mutation testing is a method used to evaluate the effectiveness of test cases by introducing small changes, known as mutants, into the program code. This technique helps in measuring the adequacy of the test cases and in generating new test cases.

A mutant is a slightly mutated version of a given program and successfully compiled. Each mutant represents a specific fault that a test case can potentially detect, it may be dependent on different programming languages.

Mutation operators define specific transformation rules to generate mutants. These operators change certain aspects of the code, such as arithmetic operations or conditional statements. Here are some mutation operators:

i. **Arithmetic Operator Replacement**: replace one arithmetic operation with another.

a) Original program: C = A+B

b) Mutants:
- C = A - B (replace + with -)

- C = A * B (replace + with *)
- C = A / B (replace + with /)
- C = A + 1 (replace B with 1)

ii. **Relational Operator Replacement**: swaps relational operators to create a mutant.

a) original program: if (A > B)

b) Mutants:

- if (A < B) (replace > with <)
- if (A >= B) (replace > with >=)
- if (A <= B) (replace > with <=)
- if (A == B) (replace > with ==)

iii. **Logical Operator Replacement**: changes logical operators within boolean expressions.

a) original program: if (A && B)

b) Mutants:

- if (A || B) (replace && with ||)
- if (!A && B) (negate A and keep B)
- if (A && !B) (negate B and keep A)
- if (!A || !B) (negate both A and B)

iv. **Variable Replacement**: substitutes one variable for another.

a) original program: C = A+B

b) Mutants:

- C = D + B (replace A with D)
- C = A + D (replace B with D)
- C = D + E (replace both A and B with other variables D and E)
- C = A + 0 (replace B with 0)

v. **Control Statement Replacement**: alters control structures to test different program flows.

a) original program:
```
if (condition) {
    // do smt
}
```

b) Mutants:

- while (condition) {/* do smt */} (replace if with while)
- for (int i = 0; i < N; i++) {/* do smt */} (replace if with for)
- if (!condition) { /* do smt */ } (negate condition)
- switch (variable) {case value: /* do smt */} (change control structure)

A mutant is considered "killed" when a test case produces a different output for the mutant than it does for the original program. If the test case detects that the output differs, it confirms that the mutant can be detected. The effectiveness of a mutation testing campaign is often indicated by the mutation score, which is the ratio of killed mutants to the total number of mutants.

Task 2: Checking Leap Year Program

Github link: https://github.com/Notbeniz/LeapYear

For this task, I chose a leap-year-checking program which is from Github. I will demonstrate metamorphic testing and evaluate my testing using mutation testing below.

Original program:

```
def is_leap(year):
    leap = False

    if year % 4 == 0 and year % 100 != 0:
        leap  =  True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Figure 1. original program

The logic of this program is simply checking whether the entered year is

- It is divisible by 4 and not divisible by 100
- It is divisible by 400

If the year satisfies one in two condition, the program returns “True” which indicates it is a leap year. In contrast, the program returns “False” indicating it is not a leap year.

i. Metamorphic Relations

Before demonstrating two Metamorphic Relations and their test groups, I created two sample non-equivalent mutants to compare with the original program and prove the effectiveness of these MRs in generating test cases and testing program.

For the calculating the effectiveness I will use this formula from Lecture 12.

Effectiveness of an MR:

ratio of violations = number of violations /
(total number of violations and satisfactions)

Figure 2. Effevtiveness of an MR formula

- 1st sample mutant

Description: Instead of checking if a year divisible by 4 is not divisible by 100 for a leap year, I replace “!=” with “==” so that the program sets leap to True if the year is both divisible by 4 and divisible by 100.

```
def is_leap(year):
    leap = False

    if year % 4 == 0 and year % 100 == 0:
        leap  =  True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Figure 3.1st sample mutant

- 2nd sample mutant

Description: This mutant alters the leap year conditions to check divisibility by 3 instead of 4, showing that a year is a leap year if it is divisible by both 3 and 100.

```
def is_leap(year):
    leap = False

    if year % 3 == 0 and year % 100 == 0:
        leap  =  True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Figure 3.2nd sample mutant

a) Metamorphic relation 1: Additive Relation

Test group 1:

Description: If the year X is a leap Year, then X+400 is also a leap year.

Intuition: Since X is a leap year, adding 400 to it results in X + 400, which is guaranteed to be a leap year as all multiples of 400 are defined as leap years.

SI (Source Input): 2000

FI (Follow-up Input): 2400

FO (Source Output): True

FO (Follow-up Output): True

This logic also applies with X+800, X+1200, X+1600, X+2000 in test group 2,3,4,5 respectively.

Test Groups ID	Mutant	Source Input Follow-up Input	Expected Output	Actual Output	Status
1	Sample 1	2000	True	True	Satisfied
		2400	True	True	
	Sample 2	2000	True	True	Satisfied
		2400	True	True	
2	Sample 1	1900	False	True	Violated
		2700	False	True	
	Sample 2	1900	False	False	Violated
		2700	False	True	
3	Sample 1	1500	False	True	Violated
		2700	False	True	
	Sample 2	1500	False	True	Violated
		2700	False	False	
4	Sample 1	2100	False	True	Violated
		3700	False	True	
	Sample 2	2100	False	True	Violated

		3700	False	False	
5	Sample 1	1100	False	True	Violated
		3100	False	True	
	Sample 2	1100	False	False	Satisfied
		3100	False	False	

Table 2. Additive Relation

Conclusion: The total number of non-equivalent mutants is 2. The above table shows that 4 test groups (2,3,4,5) killed two mutants. Hence, the FDR rate of this metamorphic relation (mutation score) with these test cases is 100%. This MR is effective.

b) Metamorphic relation 2: Multiplicative Relation

Test group 1:
Description: If the year X is not leap Year, then X*3 is also not a leap year.
Intuition: A year is considered as a leap year based on specific conditions related to its divisibility by 4, 100, and 400. If X fails to meet these criteria, multiplying it by 3 will not introduce any new conditions that could classify it as a leap year. Hence, X*3 will retain the property of being a non-leap year.
SI (Source Input): 1100
FI (Follow-up Input):3300
FO (Source Output): False
FO (Follow-up Output): False
This logic also applies with X*5, X*7, X*9, X*1 in test group 2,3,4,5 respectively.

Test Groups ID	Mutant	Test Groups Values	Expected Output	Actual Output	Status
1	Sample 1	1100	False	True	Violated
		3300	False	True	
	Sample 2	1100	False	False	Violated
		3300	False	True	
2	Sample 1	1900	False	True	Violated
		9500	False	True	
	Sample 2	1900	False	False	Satisfied
		9500	False	False	
3	Sample 1	450	True	True	Satisfied
		3150	True	True	
	Sample 2	450	True	False	Satisfied
		3150	True	False	
4	Sample 1	229	False	False	Satisfied
		2061	False	False	
	Sample 2	229	False	False	Satisfied
		2061	False	False	
5	Sample 1	1500	False	True	Violated
		1500	False	True	

	Sample 2	1500	False	True	Violated
		1500	False	True	

Table 3. Multiplicative Relation

Conclusion: The total number of non-equivalent mutants is 2. The above table shows that 3 test groups (1,2,5) killed two mutants. Hence, the FDR rate of this metamorphic relation (mutation score) with these test cases is 100%. This MR is effective.

ii. Mutation Testing

For further and more precise effectiveness conclusion, I created two test cases based on those two above MRs with 30 non-equivalent test cases:

- {1200;1600;2000;2400}
- {1000;3000;9000;27000}

I then will apply 30 mutants with these two cases to determine the quality of the effectiveness of Metamorphic Relations.

1.Change Input Type

```
def is_leap(year):
    leap = False
    if year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(float(input("Enter the year: "))))
```

Description: Changed input from int to float.

2.Logical Operator Replacement

```
def is_leap(year):
    leap = False
    if year % 4 == 0 or year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Replaced and with or in the main condition.

3. Direct Return

```
def is_leap(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 400 == 0:
        return True
    return False

print(is_leap(int(input("Enter the year: "))))
```

Description: Removed the leap variable and returned directly.

4. Divisibility Change

```
def is_leap(year):
    leap = False
    if year % 3 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Replaced year % 4 with year % 3.

5. Condition Order Swap

```
def is_leap(year):
    leap = False
    if year % 400 == 0:
        leap = True
    elif year % 4 == 0 and year % 100 != 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Swapped the order of the conditions for checking century years.

6. Negation of Conditions

```
def is_leap(year):
    leap = False
    if year % 4 != 0 and year % 100 == 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Negated the leap condition.

7. Equality Replacement

```python
def is_leap(year):
    leap = False
    if year % 4 == 0 and year % 100 == 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Changed != to == in the second condition.

8. Divisibility Change

```python
def is_leap(year):
    leap = False
    if year % 5 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Changed the check from year%4 to year%5.

9. Redundant Condition Addition

```python
def is_leap(year):
    leap = False
    if year % 4 == 0 and year % 2 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Added an unnecessary check for even years.

10. Forced Leap Year

```python
def is_leap(year):
    leap = False
    if year == 2000 or year == 2400:
        leap = True
    elif year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Added specific years (2000, 2400) that are always leap years.

11. Year Decrement

```python
def is_leap(year):
    year -= 1
    leap = False
    if year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Decremented the year before evaluation.

12. Arbitrary Year Adjustment

```python
def is_leap(year):
    year += 5
    leap = False
    if year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Added 5 to the year before evaluation.

13. Specific Year Check

```python
def is_leap(year):
    leap = False
    if year % 100 == 0 and (year % 400 == 0 or year == 1900):
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Classified only specific century years as leap years.

14. Fractional Evaluation

```python
def is_leap(year):
    leap = False
    if year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    elif year % 4 == 0.5:  # Clearly incorrect logic
        leap = True
    return leap

print(is_leap(float(input("Enter the year: "))))
```

Description: Allowed for an incorrect fractional leap year check.

15. Negation in Return

```
def is_leap(year):
    leap = not False
    if year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return not leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Negated the leap status directly in return.

16. Year Adjustment Before Check

```
def is_leap(year):
    leap = False
    if (year + 1) % 4 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Added 1 to the year before the calculation.

17. Range Restriction

```
def is_leap(year):
    leap = False
    if year > 1900 and year % 4 == 0 and year % 100 != 0:
        leap = True
    elif year > 1900 and year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Restricted checks to years greater than 1900.

18. Universal or Condition

```
def is_leap(year):
    leap = False
    if year % 4 == 0 or year % 100 != 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Used or condition for all evaluations.

19. Boolean Logic Alteration

```
def is_leap(year):
    leap = False
    if not (year % 4) and (year % 100 != 0):
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Introduced not in the condition checks.

20. Truth Count Logic

```
def is_leap(year):
    leap = False
    truth_count = (year % 4 == 0) + (year % 100 != 0) + (year % 400 == 0)
    if truth_count > 1:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Counted the number of true conditions instead.

21. Evenness Check

```
def is_leap(year):
    leap = False
    if year % 2 == 0:
        leap = True if year % 100 != 0 else False
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Used evenness to distinguish leap years.

22. Divisibility Change

```
def is_leap(year):
    leap = False
    if year % 6 == 0 and year % 100 != 0:
        leap = True
    elif year % 400 == 0:
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

Description: Changed year%4 to year%6

23. Input-based Restriction

```
def is_leap(year):
    leap = False
    if year < 2000 or (year >= 2000 and year % 4 == 0 and year % 100 != 0):
        leap = True
    return leap

print(is_leap(int(input("Enter the year: "))))
```

24. Century Limitation

```
def is_leap(year):
    leap = False
    if year % 100 == 0:
        leap = True if year % 400 == 0 else False
    return leap

print(is_leap(int(input("Enter the year: "))))
```

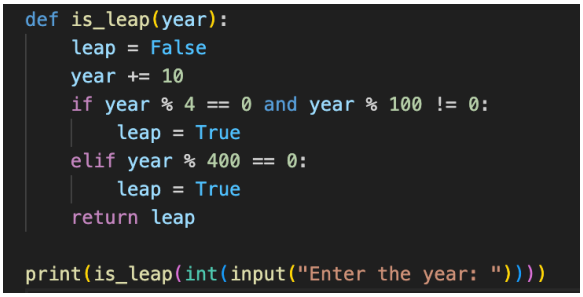
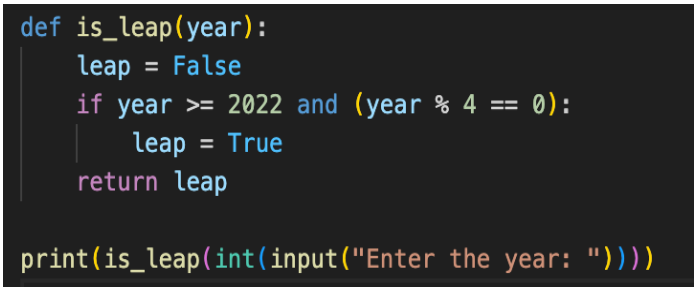
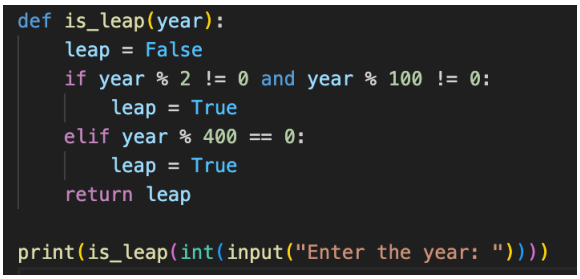
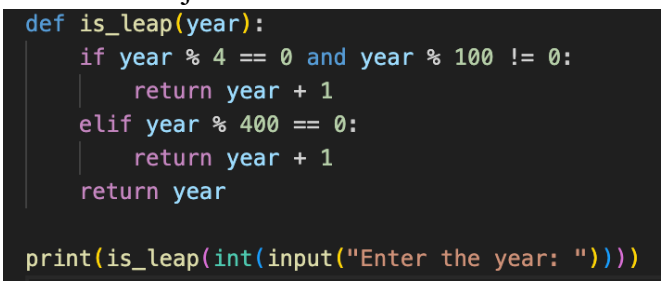
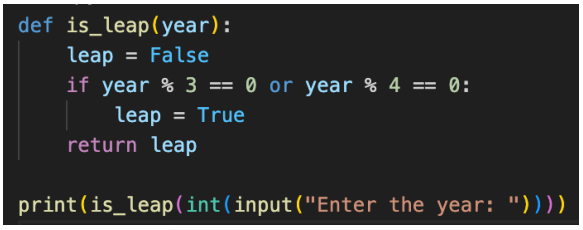
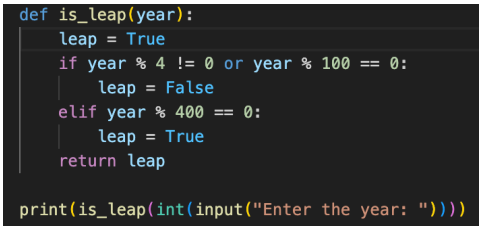
<table>
<tr><td>Description: Introduced an arbitrary limit based on input year.</td><td>Description: Limited checks to only century years.</td></tr>
<tr><td>25. Year Modification
 
Description: Adjustment of the year by adding 10 before checking.</td><td>26. Near-future Limitation
 
Description: Restricted evaluation to near-future years.</td></tr>
<tr><td>27. Odd Year Evaluation
 
Description: Evaluated odd years differently.</td><td>28. Return Adjustment
 
Description: Returns year+1 if classified as leap.</td></tr>
<tr><td>29. Divisibility Misuse
 
Description: Used divisibility by 3 to classify leap years.</td><td>30. Initial True Assumption
 
Description: Incorrectly flips boolean logic in evaluations.</td></tr>
</table>

Table 4. Mutations

Processing:
The status Green indicates mutant is not killed.
The status Red indicates that the output of the mutant and it of the original program (Expected output) does not equal, and mutant will be killed.

Test case 1: {1200;1600;2000;2400}
I created a program name testcase1.py running test case 1 with 30 mutants.
It shows 30 outputs of 30 mutants respectively.

```python
import subprocess

test_years = [1200, 1600, 2000, 2400]

for i in range(1, 31):
    mutant_file = f"../MUTANTS/mutant{i}.py"

    print(f"\nRunning tests on {mutant_file}")

    for year in test_years:
        result = subprocess.run(
            ['python', mutant_file],
            input=str(year), text=True, capture_output=True
        )
        print(f"Testing year {year} with {mutant_file}: {result.stdout.strip()}")
```

Figure 4. Test case 1 code

```
(base) dylantran@Dylans-MacBook-Air ProjectReport-104175614-VuDucTran % cd TEST
(base) dylantran@Dylans-MacBook-Air TEST % python testcase1.py

Running tests on ../MUTANTS/mutant1.py
Testing year 1200 with ../MUTANTS/mutant1.py: Enter the year: True
Testing year 1600 with ../MUTANTS/mutant1.py: Enter the year: True
Testing year 2000 with ../MUTANTS/mutant1.py: Enter the year: True
Testing year 2400 with ../MUTANTS/mutant1.py: Enter the year: True

Running tests on ../MUTANTS/mutant2.py
Testing year 1200 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 1600 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 2000 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 2400 with ../MUTANTS/mutant2.py: Enter the year: True

Running tests on ../MUTANTS/mutant3.py
Testing year 1200 with ../MUTANTS/mutant3.py: Enter the year: True
Testing year 1600 with ../MUTANTS/mutant3.py: Enter the year: True
Testing year 2000 with ../MUTANTS/mutant3.py: Enter the year: True
Testing year 2400 with ../MUTANTS/mutant3.py: Enter the year: True

Running tests on ../MUTANTS/mutant4.py
Testing year 1200 with ../MUTANTS/mutant4.py: Enter the year: True
```

Figure 5. Test case 1 output

Mutant ID	Input	Expected Output	Actual Output	Status	Mutant ID	Input	Expected Output	Actual Output	Status
1	1200	True	True		2	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
3	1200	True	True		4	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
5	1200	True	True		6	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
7	1200	True	True		8	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
9	1200	True	True		10	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
11	1200	True	False		12	1200	True	False	
	1600	True	False			1600	True	False	
	2000	True	False			2000	True	False	
	2400	True	False			2400	True	False	
13	1200	True	True		14	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	

15	1200	True	False		16	1200	True	True	
	1600	True	False			1600	True	True	
	2000	True	False			2000	True	True	
	2400	True	False			2400	True	True	
17	1200	True	False		18	1200	True	True	
	1600	True	False			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
19	1200	True	True		20	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	True			2000	True	True	
	2400	True	True			2400	True	True	
21	1200	True	False		22	1200	True	True	
	1600	True	False			1600	True	True	
	2000	True	False			2000	True	True	
	2400	True	False			2400	True	True	
23	1200	True	True		24	1200	True	True	
	1600	True	True			1600	True	True	
	2000	True	False			2000	True	True	
	2400	True	False			2400	True	True	
25	1200	True	False		26	1200	True	False	
	1600	True	False			1600	True	False	
	2000	True	False			2000	True	False	
	2400	True	False			2400	True	True	
27	1200	True	True		28	1200	True	1201	
	1600	True	True			1600	True	1601	
	2000	True	True			2000	True	2001	
	2400	True	True			2400	True	2401	
29	1200	True	True		30	1200	True	False	
	1600	True	True			1600	True	False	
	2000	True	True			2000	True	False	
	2400	True	True			2400	True	False	

Table 5. Test case 1 executation

- **Test case 2:** {1000;3000;9000;27000}

I created a program name testcase2.py running test case 2 with 30 mutants.
It shows 30 outputs of 30 mutants respectively.

```python
import subprocess

test_years = [1000, 3000, 9000, 27000]

for i in range(1, 31):
    mutant_file = f"../MUTANTS/mutant{i}.py"

    print(f"\nRunning tests on {mutant_file}")

    for year in test_years:
        result = subprocess.run(
            ['python', mutant_file],
            input=str(year), text=True, capture_output=True
        )
        print(f"Testing year {year} with {mutant_file}: {result.stdout.strip()}")
```

Figure 6. Test case 2 code

```
(base) dylantran@Dylans-MacBook-Air TEST % python testcase2.py

Running tests on ../MUTANTS/mutant1.py
Testing year 1000 with ../MUTANTS/mutant1.py: Enter the year: False
Testing year 3000 with ../MUTANTS/mutant1.py: Enter the year: False
Testing year 9000 with ../MUTANTS/mutant1.py: Enter the year: False
Testing year 27000 with ../MUTANTS/mutant1.py: Enter the year: False

Running tests on ../MUTANTS/mutant2.py
Testing year 1000 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 3000 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 9000 with ../MUTANTS/mutant2.py: Enter the year: True
Testing year 27000 with ../MUTANTS/mutant2.py: Enter the year: True

Running tests on ../MUTANTS/mutant3.py
Testing year 1000 with ../MUTANTS/mutant3.py: Enter the year: False
Testing year 3000 with ../MUTANTS/mutant3.py: Enter the year: False
Testing year 9000 with ../MUTANTS/mutant3.py: Enter the year: False
Testing year 27000 with ../MUTANTS/mutant3.py: Enter the year: False

Running tests on ../MUTANTS/mutant4.py
Testing year 1000 with ../MUTANTS/mutant4.py: Enter the year: False
Testing year 3000 with ../MUTANTS/mutant4.py: Enter the year: False
Testing year 9000 with ../MUTANTS/mutant4.py: Enter the year: False
```

Figure 7. Test case 2 ouput

Mutant ID	Input	Expected Output	Actual Output	Status	Mutant ID	Input	Expected Output	Actual Output	Status
1	1000	False	False		2	1000	False	True	
	3000	False	False			3000	False	True	
	9000	False	False			9000	False	True	
	27000	False	False			27000	False	True	
3	1000	False	False		4	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
5	1000	False	False		6	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
7	1000	False	True		8	1000	False	False	
	3000	False	True			3000	False	False	
	9000	False	True			9000	False	False	
	27000	False	True			27000	False	False	
9	1000	False	False		10	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
11	1000	False	False		12	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
13	1000	False	False		14	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	

	27000	False	False			27000	False	False	
15	1000	False	False		16	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
17	1000	False	False		18	1000	False	True	
	3000	False	False			3000	False	True	
	9000	False	True			9000	False	True	
	27000	False	True			27000	False	True	
19	1000	False	False		20	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
21	1000	False	False		22	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
23	1000	False	True		24	1000	False	False	
	3000	False	False			3000	False	False	
	9000	False	False			9000	False	False	
	27000	False	False			27000	False	False	
25	1000	False	False		26	1000	False	False	
	3000	False	False			3000	False	True	
	9000	False	False			9000	False	True	
	27000	False	False			27000	False	True	
27	1000	False	False		28	1000	False	1000	
	3000	False	False			3000	False	3000	
	9000	False	False			9000	False	9000	
	27000	False	False			27000	False	27000	
29	1000	False	True		30	1000	False	False	
	3000	False	True			3000	False	False	
	9000	False	True			9000	False	False	
	27000	False	True			27000	False	False	

Table 6. Test case 2 executation

The total number of non-equivalent mutations is 30.

Table 1: kills 9 mutations {12; 15; 17; 21; 23; 25; 26; 28; 30}

- ⇨ Mutation Score: 9/30 (0,3 out of 1)

Table 2: kills 8 mutations {2; 7; 17; 18; 23; 26; 28; 29}

- ⇨ Mutation Score: 8/30 (0,27 out of 1)

The initial table presented a mutation score of 8 out of 30, equating to 26.7%. This score implies that a considerable number of mutants, specifically 22 out of 30, went undetected,

highlighting potential areas for enhancement in the test cases to bolster fault detection capabilities.

In the subsequent table, a mutation score of 9 out of 30, also at 30%, was recorded. This finding further emphasizes the difficulties faced in achieving comprehensive fault coverage. Although the results reflect a moderate level of effectiveness, they indicate a pressing is needed for more thorough testing methods, which may involve refining existing test cases or incorporating additional mutation resources. Besides, the slight variation in scores between the two datasets may suggest inconsistencies in the performance of certain MRs or the inherent complexity associated with specific mutants.

Despite the relatively low mutation scores, the utilization of MRs was instrumental in uncovering the program's behaviour across different scenarios.

In summary, the results from the mutation testing reveal that while MRs have positively influenced fault detection, there remains significant potential for improvement in the design of test cases. The identification of 9 and subsequently 8 mutants underscore the critical role of MRs as a reliable strategy in this analysis.