Name: Vu Ha Phuong
Student ID: 104177306

FINAL REPORT

TASK 1

Subtask 1.1: Effectiveness metrics and how to apply them to evaluate random testing and partition testing.

Effectiveness metrics are measures used to assess the success or effectiveness of a particular process, system, or activity. In software testing, effectiveness metrics are used to determine the success of testing activities, ensuring performance, requirements, and quality of software. In this report, we will consider three effectiveness metrics, including P – measure, E – measure, and F – measure, and how they are used to evaluate the effectiveness of two testing processes: Random Testing and Partition Testing.

Each metric requires different input data, which in this report, we will define using distinct symbols.

Input name	Notation
Input domain	D
Domain size	d
Number of failure-causing inputs	m
Number of test cases	n
Number of partitions	k
Failure rate	$\alpha = {}^{m}/_{d}$
Sampling rate	$\beta = {}^{n}/_{d}$

1. **P – measure**

   P – measure, also known as the probability measure, is a metric that measures the likelihood of test cases occurring or the probability of errors happening when running a program. Test cases with a higher probability of finding errors are prioritized for testing. By using the P – measure to calculate the probability of error detection, it helps the tester evaluate the error detection capability in a series of test cases. This allows the tester to allocate resources more efficiently, make decisions about software quality or test coverage, and focus on parts of the system that have a higher likelihood of errors, rather than randomly testing the entire system.

   For random testing: The P – measure in random testing represents the probability of a random test case being able to detect an error.

The formula for P – measure	
With one test case	$P_r = \alpha$
With multiple test case	$P_r = 1 - (1 - \alpha)^n$

Name: Vu Ha Phuong
Student ID: 104177306

For partition testing: The P – measure in partition testing will vary for each partition. Partitions with a higher P – measure value will be prioritized for more testing, while partitions with a lower P – measure value will be tested less frequently. This helps optimize the testing process and increases the effectiveness of error detection. In partition testing, the formula will be calculated for each partition.

The formula for P – measure	
With one test case	$P_p = 1 - \prod_{i=1}^{k}(1-\alpha)$
With multiple test case	$P_p = 1 - \prod_{i=1}^{k}(1-\alpha)^{n_i}$

2. E – measure

E – measure, also known as the expected number of failures, is a metric used to calculate the expected number of errors or failures that may occur in a system or program when running test sets. It is often used to assess the risk or reliability of a software system. By using the E – measure, testers can estimate how many errors can be detected when executing a certain number of test cases. This is very useful for evaluating the effectiveness of a testing strategy and planning the testing process, especially when testing resources are limited.

For random testing: In random testing, the E – measure represents the expected number of errors based on random test cases. Unlike the P – measure, the E – measure indicates the expected number of errors when performing a certain number of random test cases, **n**.

The formula for E – measure	
With one test case	$E_r = \alpha$
With multiple test case	$E_r = n * \alpha$

For partition testing: In partition testing, each partition has its own error detection probability. The E – measure is calculated based on the number of test cases executed in each partition and the error detection probability of each partition.

The formula for E – measure	
With one test case	$E_p = \sum_{i=1}^{k} \alpha_i$
With multiple test case	$E_p = \sum_{i=1}^{k} n_i \alpha_i$

3. F – measure

The F – measure in the context of software testing is a metric that measures the expected number of test cases required to detect the first error in a series of test cases. It helps testers understand the effectiveness of the testing method and the ability to detect errors early. The F – measure is used to evaluate the efficiency of a test suite by calculating the number of test cases needed to

discover the first error in the software system. Using the F – measure can facilitate comparisons of the effectiveness of different testing strategies.

For Random Testing: When using the F – measure, you can estimate the number of test cases needed to detect the first error. The F – measure for random testing is often high due to the suboptimal probability of error detection.

The formula for the F – measure	$F_r = 1/\alpha$

For partition testing: When using the F – measure, the value of the F – measure will depend on the error detection probability of the partition being tested.

The formula for the F – measure	
With one test case	$F_p = \sum_{i=1}^{k} \beta i / \alpha\, i$
With multiple test case	$F_p = \sum_{i=1}^{k} 1/\alpha_i$

4. Using example to demonstrate the effectiveness of random testing and partition testing.

A program with an input domain consisting of 3 partitions of sizes 100, 200 and 250. There are 3, 5, and 2 failure-causing inputs, respectively in these partitons.

Partition ID	**Domain size (d)**	**Number of failure-causing inputs (m)**	**Failure rate for each partitions (α)**
P1	$d_1 = 100$	$m_1 = 3$	3 / 100 = 0.03
P2	$d_2 = 200$	$m_2 = 5$	5 / 200 = 0.025
P3	$d_3 = 250$	$m_3 = 2$	2 / 250 = 0.008

⇨ Total Domain Size (d) = 100 + 200 + 250 = 550

⇨ Total Failure-causing inputs (m) = 3 + 5 + 2 = 10

⇨ Failure rate (α) = 10 / 550

Random Testing: In Random Testing, test cases are selected randomly from the entire input domain without considering the partitioning of the partitions. Assuming in this case, we will randomly select 5 test cases. Thus, the expected number of failures (E – measure) and the probability of detecting at least one failure during testing (P – measure) will be:

$$E_r = 55 * 10 / 550 = 1$$

$$P_r = 1 – (1 – 10 / 550)^{55} = 0.6354$$

Name: Vu Ha Phuong
Student ID: 104177306

Partition Testing: In partition testing, the input domain is divided into specific partitions, and test cases are selected from each partition. Assuming in this case, a random 55 test cases will be selected and distribute to each partition without considering about the partition size.

Partition ID	Number of test cases
P1	10
P2	15
P3	30

Thus, the value of E – measure and P – measure when performing Partition Testing is:

$$E_p = (10 * 0.03) + (15 * 0.025) + (30 * 0.008) = 0.915$$

$$P_p = 1 – [(1 – 0.03)^{15} * (1 – 0.025)^{10} * (1 – 0.008)^{30}] = 0.6136$$

From the result above, we can see that the effectiveness of partition testing is lower than using random testing. To enhance the effectiveness when performing partition testing, it is necessary to use a **proportional sampling strategy (PSS)**.

Proportional sampling strategy is a testing strategy commonly used in partition testing. This strategy ensures that each partition is selected for test cases in proportion to the size of that partition, in another word, it means that they have the same sampling rate. Thus, each partition will have the following number of test cases:

Partition	Number of test cases in each partitions based on assumption	Sampling Rate (β)
P1	100 / 550 * 55 = 10 => $n_1 = 10$	B = 10 / 100 = 0.1
P2	200 / 550 * 55 = 20 => $n_2 = 20$	β = 20 / 200 = 0.1
P3	250 / 550 * 55 = 25 => $n_3 = 25$	β = 25 / 250 = 0.1

Thus, the value of E – measure and P – measure when performing Partition Testing with PSS is:

$$E_p = (10 * 0.03) + (20 * 0.025) + (25 * 0.008) = 1$$

$$P_p = 1 – [(1 – 0.03)^{10} * (1 – 0.025)^{20} * (1 – 0.008)^{25}] = 0.6364$$

Comparison of Effectiveness: Both the E-measure and P-measure show that, with the same number of test cases, Partition Testing without PSS is less effective than Random Testing. However, using PSS in Partition Testing increases its effectiveness, resulting in the increase of the value of E – measure and P – measure. When compare to Random Testing, even though the result of E – measure is the same, but P-measure is higher, making Partitiong Testing to be more efficient at detecting errors than Random Testing. In Random Testing, test cases are chosen randomly from the entire input domain, often favoring larger partitions like P3 without considering failure rates. Partition Testing with PSS, on the other hand, allocates test cases proportionally, ensuring better

coverage and higher error detection in partitions with higher failure rates, making it more effective than Random Testing.

Subtask 1.2: Metamorphic testing.

1. **Definition of Metamophic testing**

   Metamorphic testing (MT) is an effective software testing technique used when there is no test oracle to determine whether the result of a test case is correct or incorrect. Instead of relying on the exact result of each individual test case, Metamorphic Testing focuses on checking the metamorphic relations between the inputs and outputs of the program.

2. **Test Oracle**

   A Test Oracle is a mechanism or process used to determine whether the output of a software program, for a specific input, is correct or not. A Test Oracle is the only tool that helps determine whether a program is functioning correctly during testing, it allows for the verification of the results of the software testing process to see if the program behaves as expected. Without an oracle, it is impossible to know if the output is accurate, regardless of how many test cases are executed. Some common types of test oracles include Perfect Oracle, Metamorphic Oracle, Statistical Oracle, and others.

3. **Untestable Systems**

   Unstable Systems are those systems or software that cannot operate effectively or reliably. There are many reasons that could make a system untestable. Firstable, a program is untestable if there is no test oracle. A system without a test oracle to verify the correctness of results becomes a system that cannot be tested. Secondly, without clear and specific requirements, the program becomes difficult to determine the conditions for success or failure of the system. More important, in systems where the output depends on many random or indeterminate factors, verifying the results becomes challenging and leads to an unpredictable output, which can also make the system to be untestable.

4. **Motivation and Intuition of Metamorphic testing**

   4.1. **Motivation of Metamorphic testing**

   Testing programs without a test oracle: Many complex programs, such as machine learning models, make it difficult to determine the correct output for a given set of inputs. This challenges the application of traditional testing methods (like white-box and black-box testing). Metamorphic testing was developed to address this issue by focusing on the metamorphic relations between input and output sets rather than solely relying on verifying correct or incorrect results.

   More effective error detection during testing: Many systems have large or complex inputs that make it impractical to cover all possible test cases. Metamorphic testing helps detect errors based on the relationships between inputs and outputs.

Easily applicable to various types of programs: Metamorphic testing can be applied to many different types of systems and programs, especially those that receive multiple inputs and produce a large set of outputs, or untestable systems such as AI and machine learning.

4.2. Intuition of Metamorphic testing

The intuition behind metamorphic testing is based on the idea that even if the exact expected result for a specific input is unknown, testers can still rely on the relationships between multiple inputs and outputs to detect errors. These metamorphic relations are properties that the program's output must adhere to when the inputs Replace. Instead of testing the result of just one test case, metamorphic testing examines the relationships between the inputs and outputs of the program, thereby uncovering errors based on the Replaces in these properties.

5. Metamorphic Relation (MRs)

In metamorphic testing, **metamorphic relations** are concepts that describe the expected relationships between a set of inputs and outputs of a program. Unlike white-box and black-box testing, which focus on verifying the correctness of the output for a specific set of inputs, metamorphic testing emphasizes checking whether the relationships between different sets of inputs are maintained as expected. For example, a program calculates the square of a set of integers. The program takes a string of integers as input and returns the result of squaring the sum of the set. Suppose the selected input is [2, 5, 3], we will have 3 MRs as below:

ID	Description	Metamorphic Relation	Source Input	Follow Input	Source Output	Follow Output
MR1	Multiply all elements with a constant *k*	Sum of the set increase k times, square of set increase k^2 times.	[2,5,3]	[2*3, 5*3, 3*3] = [6,15,9]	$(2+5+3)^2$ = 100	$(6+15+9)^2$ = 900
MR2	Add a new element to the set	The sum of the set need to increase when a new element is added	[2,5,3]	[2,5,3,1]	$(2+5+3)^2$ = 100	$(2+5+3+1)^2$ = 121
MR3	Split the set into two separate sets	From the original set, split the set into two sets. The sum of set stay the same.	[2, 5, 3]	[2, 5], [3]	$(2+3+5)^2$ = 100	$[(2+5)+ 3]^2$ = 100

Justification:

ID	Expected Output	Verify MR	Evaluate
MR1	FO = SO * k^2	900 = 100 * 3^2	Satisfied
MR2	FO > SO	121 > 100	Satisfied
MR3	FO = SO	100 = 100	Satisfied

Name: Vu Ha Phuong
Student ID: 104177306

6. The process of Metamorphic testing and some applications

6.1. The process of Metamorphic testing consists of 7 steps.

STEP	Description
1. Identify Source Input (SI)	Provide the initial input set to be used for testing
2. Execute Testing with Source Input	Run the program using the SI and obtain the Source Output (initial result)
3. Identify Metamorphic Relations(MRs)	Define the relationships between inputs and outputs that should hold true when the input is modified
4. Transform Source Input	Modify the SI based on the MRs to create Follow-up Input (FI)
5. Execute Testing with Follow Input	Run the program with FI and obtain the Follow-up Output (new result)
6. Compare Souce Output (SO) and Follow Output (FO)	Compare SO and FO to see if the outputs meet the expected behavior according to MRs.
7. Evaluate Results	If the outputs do not satfisfy the MRs, a defect might be present in the program.

6.2. Some successful industry applications.

Metamorphic Testing has been applied in many real-world applications, especially for programs that lack specific test oracles.

Facial Recognition Systems: This is a complex program often used in security applications, most notably the facial recognition system in smartphones (like the iPhone). Testing facial recognition models is challenging because it's difficult to determine the correct output and identify a specific test oracle. Therefore, Metamorphic Testing has been applied to the testing process of facial recognition systems.

Weather Forecasting Applications: Metamorphic Testing has been effectively applied in testing weather forecasting applications, such as the Weather app on the iPhone. For an application that receives multiple inputs and where it is challenging to accurately determine the output, like weather forecasting apps, MT is a suitable testing method.

Translation Applications: A practical example is Google Translate. Translation applications will receive a complex set of inputs, making it difficult to determine the test oracle and establish the output. MT will be an appropriate testing method to help detect errors and improve the quality of translations.

Subtask 1.3: Mutation testing.

Mutation testing is a software testing method aimed at evaluating the effectiveness of a test suite by creating mutants of the original source code. The mutants are generated by making small Replaces, such as altering operators, constant values, or conditions in statements, etc.

Name: Vu Ha Phuong
Student ID: 104177306

1. Mutants

In mutation testing, mutants are modified versions of the original program. The purpose of creating mutants is to test whether the test suite is strong enough to detect these faults. If the test cases can detect the mutants, it indicates that the test cases are robust and capable of finding bugs in the original code. Conversely, if mutants are not detected, it may be necessary to improve the test suite to enhance its fault detection capability.

2. Mutation Operators

Mutation Operators are rules used in mutation testing to generate mutants. Common mutation operators include:

Mutations Operator	Mutations ID	Original program	Mutants
Replace of Relational Operator	M1	x > y	x <= y
	M2	x == y	x != y
	M3	x >= y	x < y
	M4	x != 10	x == 10
Replace of Logical Operators	M5	x && y	x || y
	M6	x || y	x && y
	M7	!(x > y)	(x > y)
	M8	!x	x
Replace variables by constants	M9	x = x + y	x = x + 1
	M10	x == y	x == 1
Constant replacement	M11	x = x * 2	x = x * 3
Replacement of logical variables by Boolean constant	M12	x && y	x && TRUE
	M13	x == y	x == FALSE
Replace of Arithmetic Operator	M14	x + y	x – y
	M15	x * y	x / y
	M16	x – y	x * y
	M17	x / y	x + y
Array reference replacement	M18	x = y[3]	x = y[4]
Assignment operator replacement	M19	x += 5	x -= 5
Scalar variable replacement	M20	x = y + z	x = y + a

3. Killed mutants

A mutant is considered "killed" when a test case is executed and it produces a different result from the original program. This indicates that the test case has the ability to detect faults present in the program. A simple example of a mutant being killed involves a program that calculates the sum of x and y.

Original program	Mutant	Test case	Expected output	Mutant output
x + y	x * y	x = 3; y = 2	3 + 5 = 8	3 * 2 = 6

Name: Vu Ha Phuong
Student ID: 104177306

Justification: The output of mutant expression is different with the output from the original program. The test case has successfully detected the errors, in other word, we could say that the mutant has been killed.

TASK 2: TESTING A PROGRAM USING MUTATION TESTING

The selected program for testing is a Python program that sort the list of numbers in ascending order using odd even sort algorithm. Link to access the program on Github: odd_even_sort.py. Using Metamorphic Testing to test the program, we will identify the Metamorphic Relations between the inputs and outputs of the program.

ID	Metamorphic Relations	Relation	Source Input	Source Output	Follow-up Input	Follow-up Output
MR1	Reorder the set	If the set is reorder, the final sorted output should remain the same (FO = SO)	[5,10,3]	[3,5,10]	[10, 5, 3]	[3, 5, 10]
			[7,15,8]	[7,8,15]	[15,7,8]	[7,8,15]
			[8,20,11,9]	[8,9,11,20]	[20, 11, 8, 9]	[8, 9, 11, 20]
			[4,1,3]	[1,3,4]	[3, 1, 4]	[1, 3, 4]
			[4,8,14,3]	[3, 4, 8, 14]	[14, 3, 4, 8]	[3, 4, 8, 11]
MR2	Add value to each elements	If a constant value *k* is added to each element of a list, the result should match sorting the list first and then adding *k* to each element. (FO = SO + k)	[7,11,1]	[1, 7, 11]	*k* = 2; [7+2,11+2, 1+2]	[3, 9, 13]
			[6,2,5,4]	[2, 4, 5, 6]	*k* = 1; [6+1,2+1,5+1, 4+1]	[3, 5, 6, 7]
			[3,18,20,0]	[0, 3, 18, 20]	*k* = 5; [3+5,18+5,20+5, 0+5]	[5, 8, 23, 25]
			[12,4,19, 9]	[4, 9, 12, 19]	*k* = 4; [12+4,4+4,19+4, 9+4]	[8, 13, 16, 23]
			[35,13, 22]	[13, 22, 35]	*k* = 3; [35+3,13+3, 22+3]	[16, 25, 38]

Below is 10 Metamorphic Groups generated out based on Metamorphic Relations. Each MR has 5 MTGs.

Metamorphic ID	Metamorphic Group ID	Metamorphic Group
MR1	**MTG1**	([5,10,3], [10, 5, 3])
	MTG2	([7,8,15], [15,7,8])
	MTG3	([8,20,11,9], [20, 11, 8, 9])
	MTG4	([4,1,3], [3, 1, 4])
	MTG5	([4,8,14,3], [14, 3, 4, 8])
MR2	**MTG6**	([7,11,1], [9,3,13])
	MTG7	([6,2,5,4], [7, 3, 6, 5])
	MTG8	([3,18,20,0], [8,23,25, 5])

Name: Vu Ha Phuong
Student ID: 104177306

	MTG9	([4,9,12,19], [8,13,16,23])
	MTG10	([13,22,35], [16,25,38])

Using mutation testing to test the program, we generate out 30 different non-equivalent mutants.

Mutant ID	File	Modify Line	Original code	Modification
M1	m1.py	32	if input_list[i] > **input_list[i + 1]**	if input_list[i] > **input_list[i / 1]**
M2	m2.py	31	for i in range(0, **len(input_list) - 1**, 2)	for i in range(0, **len(input_list) / 1**, 2)
M3	m3.py	32	if input_list[i] **>** input_list[i + 1]	if input_list[i] **<** input_list[i + 1]
M4	m4.py	31	for i in range(**0**, len(input_list) - 1, 2)	for i in range(**1**, len(input_list) - 1, 2)
M5	m5.py	33	input_list[i], input_list[i + 1] = input_list[i + 1], **input_list[i]**	input_list[i], input_list[i + 1] = input_list[i + 1], **input_list[is_sorted]**
M6	m6.py	41	is_sorted = **False**	is_sorted = **True**
M7	m7.py	32	if input_list[i] **>** input_list[i + 1]	if input_list[i] **==** input_list[i + 1]
M8	m8.py	31	for i in range(0, **len(input_list) - 1**, 2)	for i in range(0, **len(input_list)**, 2)
M9	m9.py	37	for i in range(**1**, len(input_list) - 1, 2)	for i in range(**0**, len(input_list) - 1, 2)
M10	m10.py	37	for i in range(1, **len(input_list) - 1**, 2)	for i in range(1, **len(input_list / 1)**, 2)
M11	m11.py	37	for i in range(0, **len(input_list) - 1**, 2)	for i in range(0, **len(input_list) - 2**, 2)
M12	m12.py	31	for i in range(0, **len(input_list) - 1**, 2)	for i in range(0, len(input_list) - **2**, 2)
M13	m13.py	29	while is_sorted is **False**	while is_sorted is **True**
M14	m14.py	42	return **input_list**	return **-input_list**
M15	m15.py	33	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[i * 1]**, input_list[i]
M16	m16.py	32	if **input_list[i]** > input_list[i + 1]	if **input_list[1]** > input_list[i + 1]
M17	m17.py	38	if **input_list[i]** > input_list[i + 1]	if **input_list[is_sorted]** > input_list[i + 1]
M18	m18.py	38	if input_list[i] **>** input_list[i + 1]	if input_list[i] **==** input_list[i + 1]
M19	m19.py	39	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[i + input[i]]** , input_list[i]
M20	m20.py	39	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[i / 1]**, input_list[i]
M21	m21.py	39	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[i - 1]**, input_list[i]
M22	m22.py	33	input_list[i], input_list[i + 1] = input_list[i + 1], **input_list[i]**	input_list[i], input_list[i + 1] = input_list[i + 1], **0**
M23	m23.py	33	input_list[i], input_list[i + 1] = input_list[i + 1], **input_list[i]**	input_list[i], input_list[i + 1] = input_list[i + 1], **input_list[3]**
M24	m24.py	33	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[4]**, input_list[i]
M25	m25.py	47	input_list = [int(x) for **x** in input().split()]	input_list = [int(x) for **-x** in input().split()]
M26	m26.py	42	return **input_list**	return **1**

Name: Vu Ha Phuong
Student ID: 104177306

M27	m27.py	33	**input_list[i]**, input_list[i + 1] = input_list[i + 1], input_list[i]	**input_list[4]**, input_list[i + 1] = input_list[i + 1], input_list[i]
M28	m28.py	33	input_list[i], input_list[i + 1] = **input_list[i + 1]**, input_list[i]	input_list[i], input_list[i + 1] = **input_list[i + 5]**, input_list[i]
M29	m29.py	32	if input_list[i] > **input_list[i + 1]**	if input_list[i] > **input_list[i * 1]**
M30	m30.py	38	if input_list[i] **>** input_list[i + 1]	if input_list[i] **<=** input_list[i+1]

The testing process consists of 3 steps. After generating all the mutants, I will test the original program with the MRs (Metamorphic Relations) to ensure all the test cases are verified. Next, I run each mutant with the same set of test cases that were pre-constructed and collect the results for both the Source Input and the Follow-up Input. Finally, I compare the Source Output with Follow-up Output and see if they satisfy the Metamorphic Relation. If the relation between Source Output and Follow-up Output is not satisfied the Metamorphic Relation, it means the mutant has been killed. Here are all links lead to screenshots output of running the test for mutants, MR1 and MR2: test_mr1, test_mr2, test_mutant_for_mr1, test_mutant_for_mr2.

Below is the summarize of output when executing all the mutants with all provided test cases

ID	MR1					MR2				
	MTG1	**MTG2**	**MTG3**	**MTG4**	**MTG5**	**MTG6**	**MTG7**	**MTG8**	**MTG9**	**MTG10**
M1	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M2	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M3	Survived	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M4	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M5	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M6	Survived	Survived	Killed	Survived	Killed	Survived	Survived	Survived	Survived	Survived
M7	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M8	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M9	Survived	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M10	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M11	Survived	Killed	Survived	Killed	Survived	Survived	Survived	Survived	Survived	Survived
M12	Survived	Survived	Survived	Survived	Killed	Survived	Survived	Survived	Survived	Survived
M13	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M14	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M15	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M16	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M17	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M18	Survived	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M19	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M20	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M21	Survived	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M22	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M23	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed

Name: Vu Ha Phuong
Student ID: 104177306

M24	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M25	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M26	Survived	Survived	Survived	Survived	Survived	Killed	Killed	Killed	Killed	Killed
M27	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M28	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed	Killed
M29	Killed	Killed	Killed	Killed	Killed	Survived	Survived	Survived	Survived	Survived
M30	Survived	Survived	Killed	Survived	Killed	Survived	Survived	Survived	Survived	Survived

Effectiveness Justification:

Metamorphic Relation	Number of survived mutants	Number of non-equivalent mutants killed	Total mutants	Mutation Score
MR1	22	128	150	128/150 = 0.8533
MR2	75	75		75/150 = 0.5

- MR1 involves reordering the Source Input list. The odd-even sorting algorithm should maintain the correct order after the set is reordered. This metamorphic relation effectively tests whether the algorithm can still achieve a sorted output despite modifications to the input order. With a score of 0.8533, MR1 proves to be effective because most mutants that mishandle the process of sorting after the reordering will be exposed. The effectiveness of this relation lies in its sensitivity to minor errors in maintaining the order when a set is reordered. Any mistakes in the sorting process will result in mutants being killed. Surviving mutants may have internal faults or handle edge cases well, such as elements that are already sorted. Thus, MR1 serves as a robust indicator of the sorting algorithm's reliability in handling reordering while still adhering to the expected sorted output.

- MR2 focuses on adding a constant *k* to each element of the list and checking if sorting behaves as expected. This relation tests whether the sorting logic remains consistent after transformation like uniformly increase the values. MR2's mutation score is **0.5,** which is approximately 0.3 lower that MR1's mutation score, indicates that while it is somewhat effective, it leaves more mutants undetected than MR1. This could be because the operation of adding a constant *k* does not always expose faults in sorting mechanism. Some mutants might still correctly sort the transformed list, even if they contain faulty behavior unrelated to transformations like constant additions.

Conclusion: MR1's higher mutation score suggests it is more effective at identifying issues related to structural changes in the input list. MR2's lower mutation score reflects that adding a constant to all elements does not challenge the sorting logic as much and may not expose all faults, making it slightly less effective. Thus, MR1 is justified to have a higher mutation score, as it tests a more sensitive aspect of the sorting process, while MR2's lower score shows that it is still useful but less robust in catching all faults.