

# CS-731 Software Testing

## Project Report

## Mutation Testing on Karatsuba Algorithm

Prem Shah (IMT2020044) and Harsh Shah (IMT2020006)

GitHub Link of the project : [https://github.com/02prem/Mutation\\_testing-PITest.git](https://github.com/02prem/Mutation_testing-PITest.git)

### Project Aim

The aim of this project is to understand and use mutation testing on a piece of code with the help of mutation tools and plugins available. The primary aim of this project is to enhance the effectiveness of the software testing process by evaluating and improving the quality of the test cases and the code it tests.

### Source Code

Link to our source code: [GitHub Link](#)

We have used the **Karatsuba Algorithm** as our source code, which we implemented previously as a course work. The Karatsuba algorithm is a fast multiplication algorithm that was discovered by Anatolii Alexeevitch Karatsuba in 1960. It is a divide-and-conquer algorithm designed to efficiently multiply two numbers. The key idea behind the algorithm is to break down the multiplication of two  $n$ -digit numbers into three multiplications of smaller-size numbers, reducing the overall number of multiplications.

A brief overview of how the Karatsuba algorithm works:

Consider multiplying two numbers, with their maximum length being  $n$ .

Steps:

1. Rewrite  $x$  and  $y$  in the form of  $n/2$  digits in the form of  $a$ ,  $b$ ,  $c$ , and  $d$ .
2. Compute the value of  $ac$ .
3. Compute the value of  $bd$ .
4. Compute the value of  $(a+b)(c+d)$
5. Calculate  $ad + bc$  by subtracting the answer of step4 — step3 — step2

6. Finally compute the answer by making a few additions/subtractions to our computed values using the below expression.

It is important to note that we are calculating  $ad + bc$ , by multiplying  $(a+b)$  and  $(c+d)$  and then performing some subtractions. This is crucial for algorithmic performance, as we know that additions/subtraction operations are faster than multiplications. It replaces the traditional four multiplications with three multiplications, reducing the overall complexity of the multiplication operation.

The code was converted to java for better and convenient mutation testing. The source code consists of about 650 lines and the testing code of about 300 lines for 35 tests. This makes of about **950-1000** lines of code.

For more testing we added some hard problems from our Leetcode submissions which consists of about 370 lines of source and 180 lines of testing of 26 test functions with about 35-40 tests. This was added to make our overall testing part over 1000+ lines of source code.

## Testing Strategy and Tools

We built a maven project for implementing the mutation testing on the source code. The mutation testing was done by using '**PIT mutation testing**' tool. It is available as a plugin which is needed to be added in the xml file and is easy to use tool. It creates multiple mutants of the source code for testing. A mutant is a variant of the source code but with slight modification in that code. In this way our test cases can either pass or fail on the mutant.

Our aim is to kill the mutant. There are two ways to kill a mutant: strongly or weakly.

**Strongly killed**: A mutant is considered strongly killed if the test suite can detect and fail the test when the mutant is introduced. They demonstrate that the test case is effective in detecting and rejecting code changes that impact program behaviour.

**Weakly killed**: A weak kill happens when the mutation does not affect the program behaviour in a way that triggers any test failure, and the test suite is not sensitive enough to identify the mutation. Having too many weak kills suggests that the test suite may have blind spots and may not be as effective in capturing all potential defects.

The **mutation score** is a metric used in mutation testing to measure the effectiveness of a test suite in detecting artificial code mutations. It provides a quantitative assessment of how well the tests can identify and "kill" these mutations. It is the proportion of mutants that are successfully killed by the test suite out of the total number of mutants introduced.

How to interpret mutation score?

- A high mutation score (close to 100%) suggests that the test suite is effective in identifying and killing mutations, indicating a robust set of tests.

- A low mutation score indicates that a significant portion of mutants were not detected by the test suite, suggesting potential weaknesses in the test coverage.

However, interpreting the results requires consideration of the specific characteristics of the mutants and the context of the software being tested. Some mutations cannot be identified because of going into unreachable state like infinite loop. The goal is to achieve a balance where the mutation score is high enough to instil confidence in the test suite's ability to catch defects.

For writing test cases we used the **Junit** java testing tool which allows us to write manual test cases and compare the test results.

## Unit and Integration Testing

Unit testing focuses on testing individual components or units of code in isolation. Each unit test verifies the correctness of a specific function or method. Integration testing verifies the interactions and interfaces between different components or modules. It ensures that these components work together as expected when integrated into the larger system.

We have implemented both unit and integration testing as a part of our project.

In mutation testing, unit testing is basically just testing all the functions or methods that are there in the code. It can be done by modifying the logical, arithmetical or comparator operators or by changing the variables. We used the following **mutation operators for unit testing**:

- CONDITIONALS\_BOUNDARY
- INCREMENTS
- MATH
- NEGATE\_CONDITIONALS

In mutation testing, integration testing involves testing the function calls. It can be done by modifying the function calls, return statements or changing/inter-changing functional parameters. We used the following **mutation operators for integration testing**:

- EMPTY\_RETURNS
- FALSE\_RETURNS
- NULL\_RETURNS
- PRIMITIVE\_RETURNS
- TRUE\_RETURNS
- VOID\_METHOD\_CALLS

## Active mutators

- CONDITIONALS\_BOUNDARY
- EMPTY\_RETURNS
- FALSE\_RETURNS
- INCREMENTS
- INVERT\_NEGS
- MATH
- NEGATE\_CONDITIONALS
- NULL\_RETURNS
- PRIMITIVE\_RETURNS
- TRUE\_RETURNS
- VOID\_METHOD\_CALLS

Figure 1. Mutators used in testing

## Mutation Testing Report

### Pit Test Coverage Report

#### Package Summary

com.mycompany.app

| Number of Classes | Line Coverage  | Mutation Coverage | Test Strength  |
|-------------------|----------------|-------------------|----------------|
| 3                 | 94%<br>598/639 | 84%<br>548/654    | 88%<br>548/625 |

#### Breakdown by Class

| Name                              | Line Coverage  | Mutation Coverage | Test Strength  |
|-----------------------------------|----------------|-------------------|----------------|
| <a href="#">Algorithms.java</a>   | 97%<br>203/210 | 83%<br>169/203    | 84%<br>169/202 |
| <a href="#">Leetcode.java</a>     | 99%<br>180/182 | 88%<br>175/198    | 89%<br>175/196 |
| <a href="#">NumberTheory.java</a> | 87%<br>215/247 | 81%<br>204/253    | 90%<br>204/227 |

Report generated by [PIT](#) 1.15.3

The above figure shows the overall PIT test coverage report. '**NumberTheory.java**' is our main source code. We have achieved 80-85 % mutation coverage in each class.

## Screenshots of Unit Testing

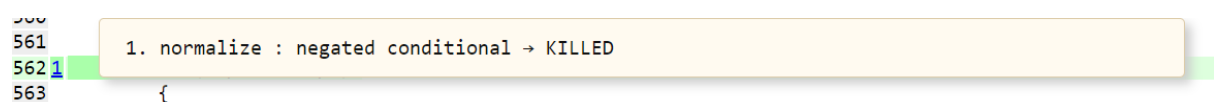


Figure 2. NEGATE\_CONDITIONALS operator on 'if' block

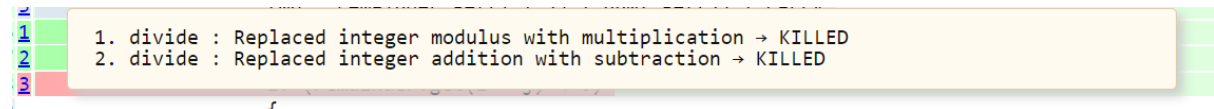


Figure 3. MATH operator applied to change the arithmetic operations

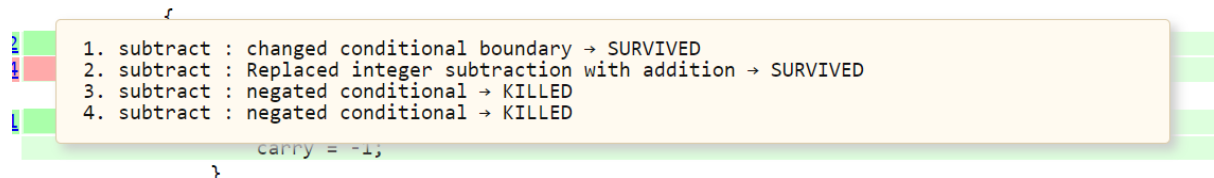


Figure 4. Multiple mutation operators applied on a single line of code

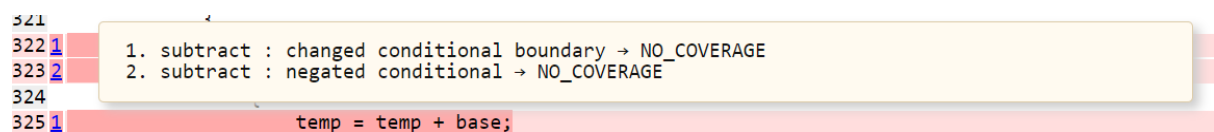


Figure 5. Mutants which are not reachable because of TIMED\_OUT suggesting that the mutant went to infinite loop

## Screenshots of Integration Testing

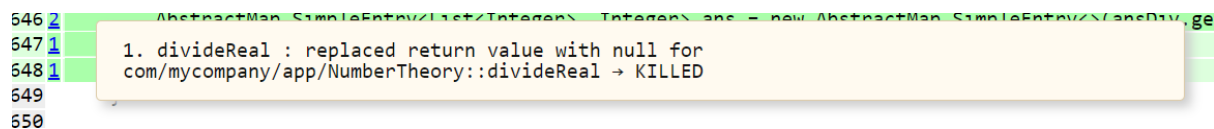


Figure 6. NULL\_RETURNS operator

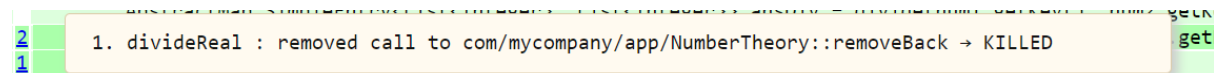


Figure 7. VOID\_METHOD\_CALLS operator

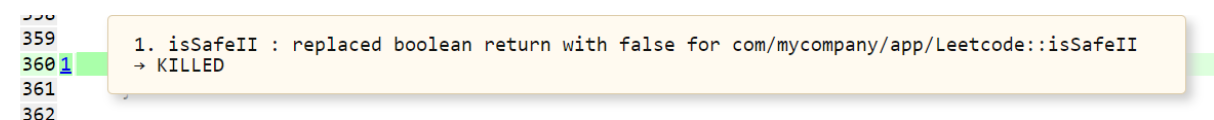


Figure 8. FALSE\_RETURNS operator

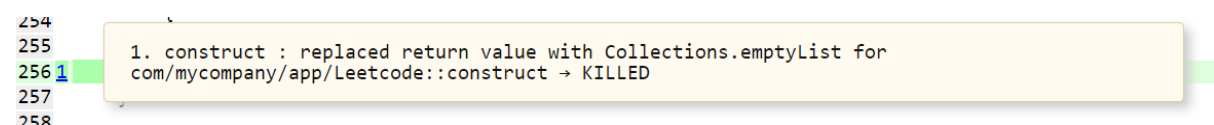
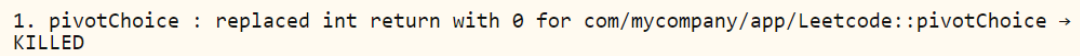


Figure 9. EMPTY\_RETURNS operator



```
1. pivotChoice : replaced int return with 0 for com/mycompany/app/Leetcode::pivotChoice -> KILLED
```

*Figure 10. PRIMITIVE\_RETURNS operator which replaces the return type with 0*