Mutation Testing on Various DSA problems

Instructor: *Prof. Meenakshi D Souza*

Team Members

Ankush Patil (Roll Number: MT2023101)
Shatakshi Tiwari (Roll Number:MT2023175)

Abstract:

This project explores the practical aspects of mutation testing on Data Structure and Algorithm (DSA) problems using two tools:

1. **PIT Mutation Testing for Java**: A mutation testing framework for Java.
2. **Stryker Mutator for JavaScript**: A mutation testing framework for JavaScript.

The primary goal was to evaluate the robustness of test cases by generating mutants, applying different mutation operators at both the **unit** and **integration** levels, and analysing the results to identify and improve weak test cases.

Introduction to Mutation Testing :

Mutation testing is a fault-based software testing technique that measures the effectiveness of test cases in detecting faults. It works by generating **mutants**, which are slightly modified versions of the original code. These modifications are introduced using **mutation operators**, such as changing logical operators (`&&` → `||`), altering arithmetic operations (`+` → `-`), or negating conditions (`==` → `!=`).

The mutants are then executed against the existing test cases to determine their quality. A test case is deemed effective if it can differentiate between the original code and its mutated version by failing for the mutated code while passing for the original.

Key Concepts in Mutation Testing:

1. **Strongly Killed Mutant:** A mutant is strongly killed if the test case detects the fault introduced in the mutant, i.e., the output differs from the expected outcome for the original code.
2. **Surviving Mutant:** A mutant is said to survive if it behaves like the original code and the test cases fail to detect the mutation. Surviving mutants indicate inadequacy in the test cases.
3. **Mutation Score:** The percentage of mutants killed by the test cases. A high mutation score suggests robust test coverage.

$$\text{Mutation Score} = \left(\frac{\text{Killed Mutants}}{\text{Total Number of Mutants}} \right) \times 100$$

Where:

- **Killed Mutants**: The number of mutants detected and eliminated by the test cases.
- **Total Number of Mutants**: The total number of mutants generated using mutation operators.

Why Mutation Testing?

- It provides a deeper evaluation of test case effectiveness compared to traditional coverage metrics like line or branch coverage.
- It identifies "untested" areas in the code that conventional techniques might overlook.
- It encourages rigorous testing practices by revealing subtle issues and edge cases in code.

DSA Problem Description:

The DSA problems selected for testing include:

Category	Function	Description
Math Utilities	factorial	Calculates the factorial of a number.
	gcd	Finds the greatest common divisor (GCD) of two numbers.
	lcm	Computes the least common multiple (LCM) of two numbers.
	fibonacci	Finds the nth Fibonacci number.
	power	Computes the power of a number using recursion.
	findDivisors	Finds all divisors of a number.
	isPerfectNumber	Checks if a number is a perfect number.
	largestPrimeFactor	Finds the largest prime factor of a number.
	sieveOfEratosthenes	Generates all prime numbers up to a given limit.

	isArmstrongNumber	Checks if a number is an Armstrong number.
DSA Algorithms	bubbleSort	Sorts an array using bubble sort.
	findMax	Finds the maximum value in an array.
	reverseString	Reverses a given string.
	isPalindrome	Checks if a string is a palindrome.
	areAnagrams	Checks if two strings are anagrams.
	binarySearch	Performs binary search on a sorted array.
	knapsack	Solves the 0/1 knapsack problem.
	bfs	Performs Breadth-First Search (BFS) traversal on a graph.
	lcs	Finds the length of the longest common subsequence between two strings.
	lis	Finds the length of the longest increasing subsequence in an array.
	prims	Finds the MST using Prim's algorithm.
	rabinKarp	Finds all occurrences of a pattern in a text using Rabin-Karp algorithm.
	KMP	Finds all occurrences of a pattern in a text using KMP algorithm.

Each problem involves complex logic and control structures, making them suitable for mutation testing.

Mutation Tools Used:

1. PIT Mutation Testing (Java):

- URL: https://pitest.org/
- Features**:**
    - Supports various mutation operators like negation, conditional boundary changes, and maths operation changes.
    - Provides detailed mutation coverage reports.

2. Stryker Mutator (JavaScript):

- URL: https://stryker-mutator.io/
- Features**:**
    - Supports JavaScript codebases with easy integration.
    - Includes operators like statement removal, boolean replacement, and arithmetic operator mutation.

Test Case Strategy:

The test cases were designed to:

- Comprehensive Code Coverage: Ensuring all paths, including edge cases and boundary conditions, are tested.
- Mutation-Targeted Design: Crafting test cases that effectively handle mutation operators applied at both unit and integration levels.
- High Mutation Coverage: Maximising the mutation kill ratio by detecting and eliminating surviving mutants.

Mutation Operators Applied:

Few of the mutation operator are listed below

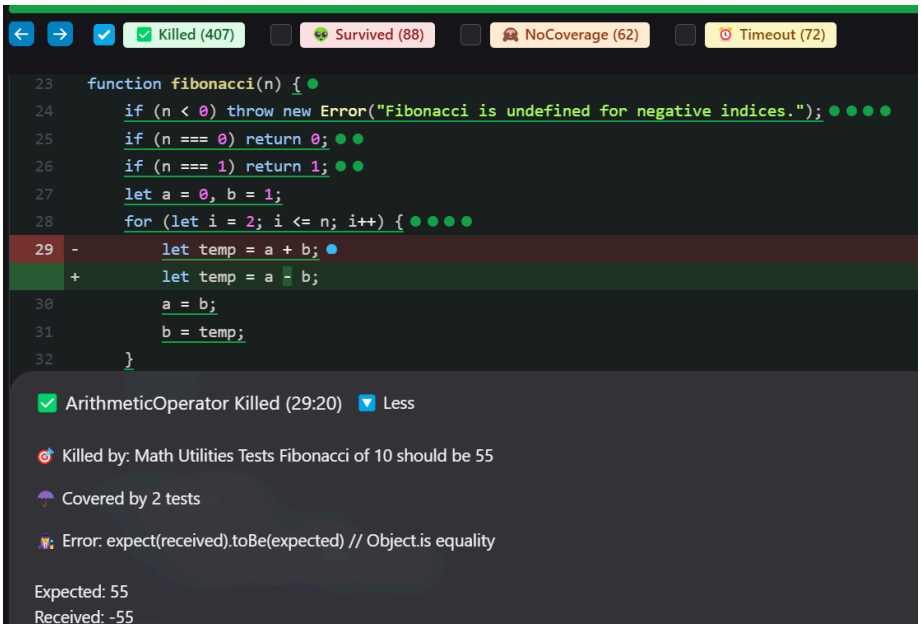
Mutation Operator	**Description**	**Example**
Arithmetic Operator Mutation	Replace arithmetic operators (+, -, *, /, %) with other	a + b → a - b

	operators.	
Relational Operator Mutation	Replace relational operators (<, >, <=, >=, ==, !=) with others.	a == b → a != b
Logical Operator Mutation	Replace logical operators (&&,||) with others.	a && b → a|| b
Conditional Operator Mutation	Alter conditions in control structures (if, while, for, etc.)	if (a > b) → if (a <= b)
Negation Mutation	Negate expressions or boolean conditions.	if (a > b) → if (!(a > b))
Constant Replacement Mutation	Replace constants or literals with other values.	return 42; → return 0;
Variable Replacement Mutation	Replace one variable with another of compatible type.	a = b; → a = c;
Code Removal Mutation	Remove specific code blocks or statements.	a++; b++; → a++;

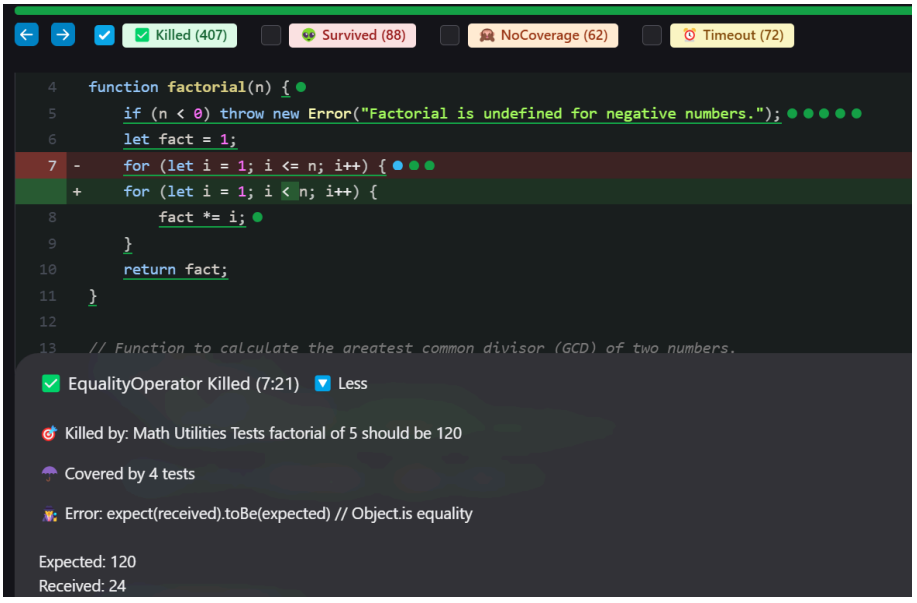
1.Stryker

Command to Run : *npx stryker run*

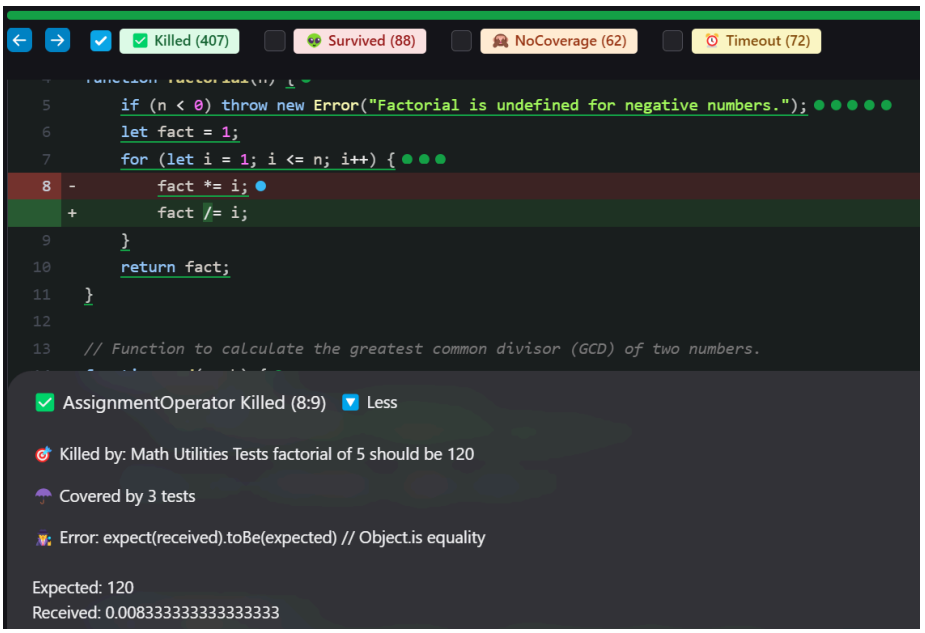
1)ArithmeticOperator Killed



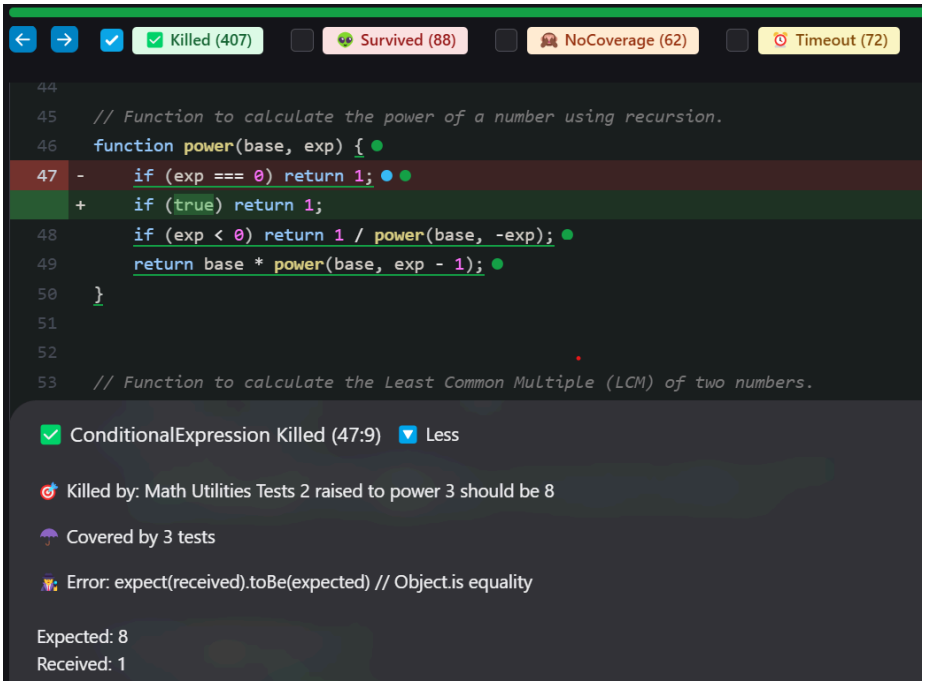
2)EqualityOperator(Relational) Killed



3)AssignmentOperator Killed



4)ConditionalExpression Killed



2.Java

Command to Run : *mvn org.pitest:pitest-maven:mutationCoverage*

```
246       /**
247        * Function to check if a string is a palindrome.
248        */
249       public static boolean isPalindrome(String str) {
250 1         int left = 0, right = str.length() - 1;
251 2         while (left < right) {
252 1             if (str.charAt(left) != str.charAt(right)) {
253 1                 return false;
254               }
255 1             left++;
256 1             right--;
257           }
258 1         return true;
259       }
260
261       /**
262        * Binary Search Algorithm.
263        */
264       public static int binarySearch(int[] arr, int target) {
265 1         int left = 0, right = arr.length - 1;
266 2         while (left <= right) {
267 3             int mid = left + (right - left) / 2;
268 2             if (arr[mid] == target) return mid;
269 3             else if (arr[mid] < target) left = mid + 1;
270 1             else right = mid - 1;
271           }
272 1         return -1;
273       }
274
```

Mutation Operators in our Code in PIT (Java)

256	1. Changed increment from -1 to 1 → KILLED
258	1. replaced boolean return with false for org/example/Main::isPalindrome →
265	1. Replaced integer subtraction with addition → KILLED
266	1. changed conditional boundary → SURVIVED 2. removed conditional - replaced comparison check with false → KILLED
267	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer division with multiplication → KILLED 3. Replaced integer addition with subtraction → KILLED
268	1. replaced int return with 0 for org/example/Main::binarySearch → KILLED 2. removed conditional - replaced equality check with false → KILLED
269	1. changed conditional boundary → SURVIVED 2. removed conditional - replaced comparison check with false → SURVIVED 3. Replaced integer addition with subtraction → TIMED_OUT
270	1. Replaced integer subtraction with addition → NO_COVERAGE
272	1. replaced int return with 0 for org/example/Main::binarySearch → KILLED
286	1. removed call to java/util/Arrays::fill → KILLED
289	1. Replaced integer subtraction with addition → KILLED 2. removed conditional - replaced comparison check with false → KILLED 3. changed conditional boundary → SURVIVED
293	1. changed conditional boundary → KILLED 2. removed conditional - replaced comparison check with false → KILLED
294	1. Replaced integer addition with subtraction → SURVIVED 2. removed conditional - replaced equality check with false → KILLED 3. removed conditional - replaced comparison check with false → KILLED 4. changed conditional boundary → SURVIVED 5. removed conditional - replaced equality check with false → KILLED 6. removed conditional - replaced equality check with false → KILLED
297	1. Replaced integer addition with subtraction → KILLED
301	1. replaced return value with null for org/example/Main::dijkstra → KILLED
309	1. changed conditional boundary → KILLED 2. removed conditional - replaced comparison check with false → KILLED
310	1. removed conditional - replaced comparison check with false → KILLED 2. changed conditional boundary → SURVIVED

Active Mutation Operators in our Code

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NULL_RETURNS
- PRIMITIVE_RETURNS
- REMOVE_CONDITIONALS_EQUAL_ELSE
- REMOVE_CONDITIONALS_ORDER_ELSE
- TRUE_RETURNS
- VOID_METHOD_CALLS

Integration Mutation Testing(in Javascript)

- Integration mutation testing focuses on verifying the interactions between multiple components or functions to ensure they work together as expected. Unlike unit testing, which isolates individual functions, integration testing checks how these functions collaborate, ensuring that the system behaves as intended when combined.
- In the context of the project, we applied integration mutation testing using several mutation operators:
    1. **IVPR (Integration Parameter Variable Replacement)**: This mutation operator modifies parameters by replacing them with alternative values. For example, we tested the relationship between Fibonacci and Factorial by replacing the parameter replacement, ensuring that their interaction holds under different conditions.

       Example: Testing the relationship between Fibonacci and Factorial results by modifying input values and ensuring correct outputs.

```
test("Integration Test: Fibonacci and Factorial IVPR", () => {
  const fibValue = fibonacci(5);
  const factValue = factorial(5);
  expect(factValue / fibValue).toBe(24);
});
```

    2. **IPEX (Integration Parameter Exchange)**: This operator swaps function parameters to test how the functions interact with different inputs. For instance, we tested GCD and LCM functions by exchanging their parameters to validate the correct output.

Example: Testing GCD and LCM functions with swapped parameters.

```
test("Integration Test: GCD and LCM IPEX", () => {
  const gcdResult = gcd(12, 18);
  const lcmResult = (12 * 18) / gcdResult;
  expect(lcmResult).toBe(36);
});
```

3. **IMCD (Integration Method Call Deletion)**: This operator removes method calls to test if the interaction between functions can still maintain the correct outcome. For example, we tested the findDivisors function alongside the isPerfectNumber function, ensuring the correct result even when intermediate steps are removed.

```
test("Integration Test: Perfect Number Validation IMCD", () => {
  const number = 496;
  const isPerfect = isPerfectNumber(number);
  expect(isPerfect).toBe(true);
});
```

4. **IREM (Integration Return Expression Modification)**: This mutation alters the return expression of a function to test if the overall workflow still produces the expected result. In our tests, we modified the return of sorting functions and checked if the final output is valid.

Example: Testing bubble sort and ensuring the array is sorted correctly after modification.

```
test("Integration Test: Bubble Sort IREM with odd-sized array", () => {
  const arr = [31, 45, 22, 90, 67, 11];
  bubbleSort(arr);
  expect(arr).toEqual([11, 22, 31, 45, 67, 90]);
});
```

Implementation Steps:

1. Setting up PIT for Java:

- Selected DSA problems implemented in Java.
- Configured PIT with Maven for the Java projects.
- Designed JUnit test cases for each problem.
- Generated mutation reports using PIT, identifying killed and surviving mutants.

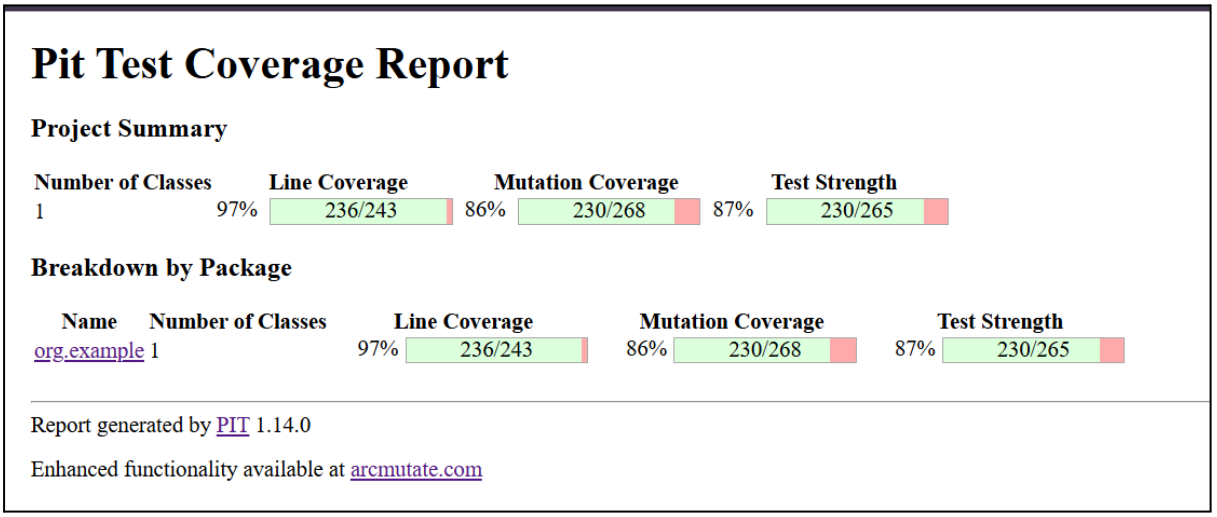
2. Setting up Stryker for JavaScript:

- Selected the JavaScript implementations of the DSA problems.
- Configured Stryker using Node.js.
- Designed Jest test cases for the problems.
- Generated mutation reports, highlighting weaknesses in the test cases.

Results:

Mutation Coverage Metrics:

PIT Mutation Testing (Java)(Overall) :



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	97% 236/243	86% 230/268	87% 230/265

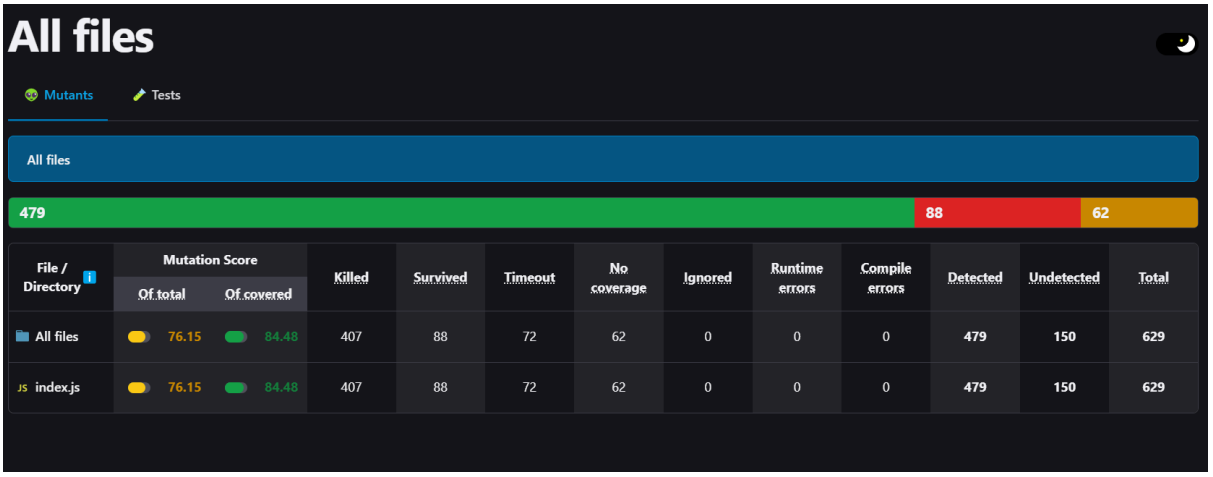
Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.example	1	97% 236/243	86% 230/268	87% 230/265

Report generated by PIT 1.14.0

Enhanced functionality available at arcmutate.com

Stryker Mutation Testing (JavaScript)(Overall) :



Comparison between PIT and Stryker

PIT Report:

- Line Coverage: **97%** of the lines in the mutated classes were covered by the tests (236/243).
- Test Strength: **87%**, which means the tests are strong enough to detect **87%** of the generated mutants.
- Generated Mutants: A total of **268 mutants** were generated, with **230 killed** and 38 survived.
- Time Taken: Total mutation analysis took **37 seconds**, with detailed breakdown of timings for each phase.

Stryker Report:

- Line Coverage: **84.48%** of the mutations were covered by the tests.
- Killed Mutants: **407 mutants** were killed, showing that the mutation tests were fairly effective, but the total number of mutants seems to differ from the PIT report's total.
- Timeouts and Errors: The Stryker report lists 72 timeouts, compared to 3 timeouts in PIT. This is significant as it may indicate performance issues with certain mutations under Stryker.
- Test Strength: The report doesn’t explicitly mention test strength, but the tests per mutation ratio is higher at **2.11 (Stryker)** compared to **1.18 (PIT)**,

indicating Stryker is running more tests per mutant, potentially making the tests more exhaustive.
- Time Taken: Total mutation analysis took **1min 17sec**

Conclusion:

PIT outperforms Stryker in terms of mutation score **(86% vs. 81.05%**) and execution time **(37 seconds vs. 1 minute 17 seconds**), making it more efficient for faster mutation testing with fewer timeouts.

Stryker runs more tests per mutant (**2.11**), which could lead to more thorough analysis but at the cost of higher timeouts (**72 vs. 3 in PIT**).

PIT is recommended for faster, more efficient mutation testing, while **Stryker** is better suited for exhaustive testing despite the longer runtime.

Team Contributions:

- **Shatakshi Tiwari:** Configured and executed **PIT** for Java, analysed mutation reports, and improved Java test cases.
- **Ankush Patil:** Configured and executed **Stryker** for JavaScript, analysed mutation reports, and improved JavaScript test cases.