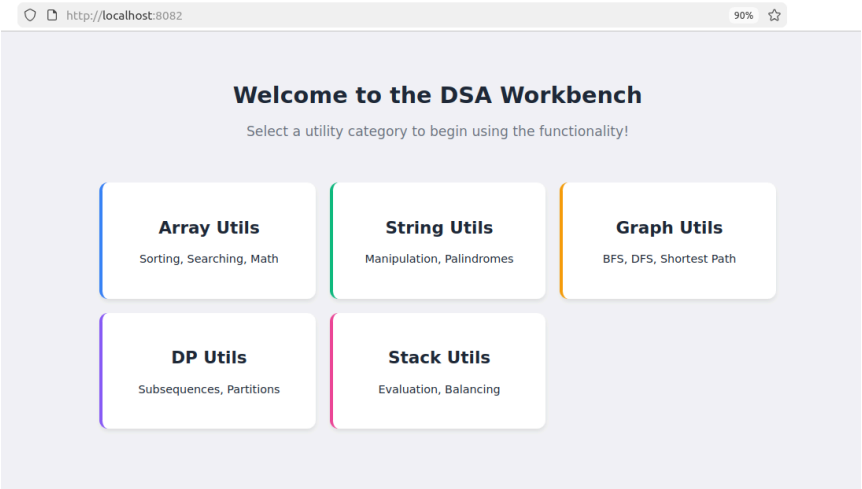
CSE 731: Software Testing Project Report

Comprehensive Validation of DSA Workbench: Mutation, E2E, and Performance Testing

By Aayush Bhargav (IMT2022089) and Praveen Peter Jay (IMT2022064)

1. Introduction

This report documents the results of a comprehensive quality assurance campaign for our **DSA Workbench** application, focusing on selected families of core Data Structures and Algorithms (DSA) logic problems. Our strategy utilized **mutation testing (Pitest)** to maximize unit test rigor, supplemented by **Selenium E2E testing** and **concurrent load testing** for system-wide validation.



The initial Pitest run highlighted significant weaknesses, revealing a low **Mutation Coverage of 63%**. Following an intensive test development phase, we successfully eliminated hundreds of generated faults, leading to exceptional improvements:

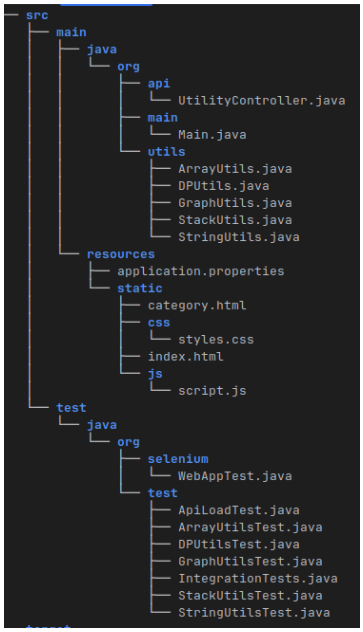
- **Mutation Coverage improved from 63% to 82%.**.
- **Test Strength improved from 75% to 85%.**
- **Line Coverage improved from 78% to 94%.**

This integrated approach confirms that the application's core logic is validated by a high-fidelity test suite, is functionally correct across the stack, and remains stable under heavy load.

2. DSA Workbench: Application Architecture

2.1 System Overview

The **DSA Workbench** is a high-performance utility application built on a **Spring Boot** backend that exposes core DSA functionalities via a REST API. Instructions to run the app are available on github.



The structure of the src directory of the project

2.2 Application Components

1. **Frontend (UI):** A simple HTML/CSS interface (index.html) allows users to select a utility category (Array, String, Graph, DP, Stack) and execute specific functions via the backend.
2. **Backend (API):** The UtilityController.java Spring Boot application serves as middleware. It receives client requests (as a generic RequestDTO), performs necessary type conversions (e.g., parsing strings into int[] or graph structures), routes the execution to the corresponding static method in the org.utils package, and returns the result encapsulated in a ResponseDTO.

This design ensures that the core business logic in org.utils is isolated, highly tested, and accessible via a robust API.

3. Core Validation: Unit Testing & Mutation Campaign

3.1 Tooling and Configuration

We utilized Pitest, a mutation testing framework integrated via Maven, to inject faults (create mutants) and measure the effectiveness of our test suite. The goal was to ensure tests not only satisfy code coverage but also verify its correctness by killing mutants.

Configuration Detail	Value	Rationale
PIT Version	1.22.0	Latest stable version for advanced features.
Mutators	ALL	Ensures maximum variety of fault-injection mechanisms, testing subtle bugs.
Target Classes	org.utils.*	Scope limited to the core utility package.
Mutation Threshold	80	Sets a mandatory minimum acceptable mutation score.

3.2 Unit Test Suite Expansion (JUnit)

The mutation campaign necessitated a massive expansion of our existing JUnit test suite to cover previously weak code paths. The org.utils package comprises five utility files, each with an analogous test file in org.test. Each function in the utility files started off with 1-2 test cases.

The initial and final test run reports demonstrate this significant growth:

Test Class	Initial Tests Run	Final Tests Run	Growth
ArrayUtilsTest	19	71	52
StringUtilsTest	15	38	23
DPUtilsTest	9	41	32
StackUtilsTest	10	22	12
GraphUtilsTest	10	20	10
Total Test Cases	**63**	**192**	**129**

The **204%** increase in test cases directly correlates with the success of the mutation campaign, ensuring robustness against edge cases and logical faults.

3.3 Metrics Defined

Metric	Definition
Line Coverage	Percentage of code lines executed by tests.

Mutation Coverage	Percentage of mutants killed out of the total generated.
Test Strength	Percentage of code lines covered by *effective* tests (tests that kill at least one mutant).

3.4 Comprehensive Mutator Strategy

Our campaign employed 26 distinct mutators from the ALL configuration, ensuring maximum fault-injection and test rigor across different code types:

I. Conditional and Flow Control Mutators

- CONDITIONALS_BOUNDARY: Tests boundary conditions by replacing < with <= (6 mutators).
- NEGATE_CONDITIONALS: Reverses logical flow by mutating operators (== with !=, etc.).
- REMOVE_CONDITIONALS_EQUAL_IF/ELSE and REMOVE_CONDITIONALS_ORDER_IF/ELSE: Removes conditional statements, forcing specific code blocks to always or never execute (4 mutators).

II. Arithmetic and Value Mutators

- MATH: Replaces binary arithmetic operators (+ with -, * with /).
- INVERT_NEGS: Inverts negation of integer and floating-point variables.
- INCREMENTS and REMOVE_INCREMENTS: Mutates or removes local variable increments.
- INLINE_CONSTS: Mutates literal values (e.g., changing 42 to 43).

III. Return Value Mutators

- EMPTY_RETURNS: Replaces collection/string returns with an "empty" value.
- PRIMITIVE_RETURNS: Replaces primitive returns with 0.
- TRUE_RETURNS, FALSE_RETURNS, NULL_RETURNS: Forces boolean and object returns to their extreme or null states (3 mutators).

IV. Method and Constructor Call Mutators

- VOID_METHOD_CALLS: Removes calls to methods that return void.
- NON_VOID_METHOD_CALLS: Removes calls to non-void methods, returning the Java default value (e.g., 0 or null).
- CONSTRUCTOR_CALLS: Replaces object construction (new Object()) with null.

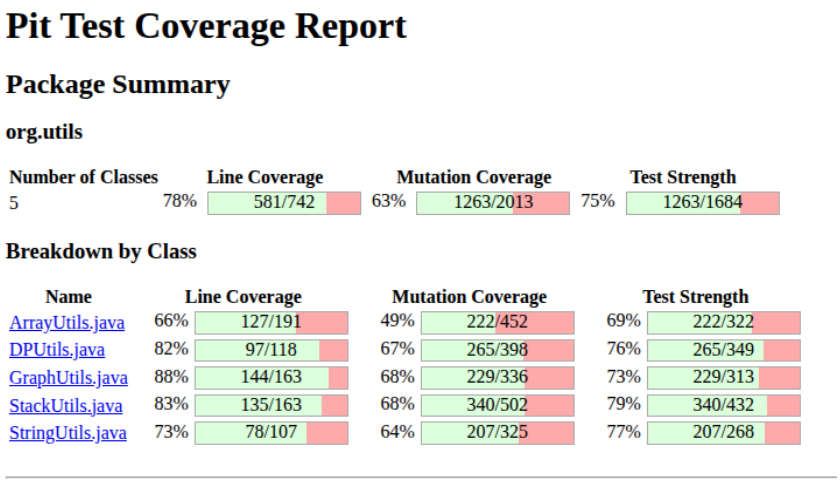
V. Experimental Mutators

- Aggressive mutators targeting complex Java constructs:

EXPERIMENTAL_ARGUMENT_PROPAGATION, EXPERIMENTAL_BIG_DECIMAL, EXPERIMENTAL_BIG_INTEGER, EXPERIMENTAL_MEMBER_VARIABLE, EXPERIMENTAL_NAKED_RECEIVER, EXPERIMENTAL_SWITCH, and EXPERIMENTAL_REMOVE_SWITCH_MUTATOR_[0-99].

4. Results Overview and Analysis

The mutation testing campaign delivered substantial and consistent improvements across all key metrics and utility classes, validating the **80%** mutation threshold.



Pit Test Coverage Report

Package Summary

org.utils

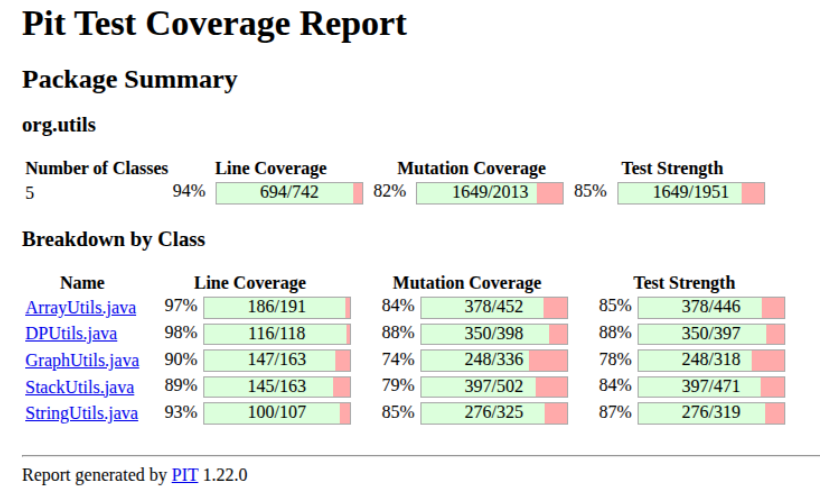
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
5	78%	581/742	63%	1263/2013	75%	1263/1684

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ArrayUtils.java	66%	127/191	49%	222/452	69%	222/322
DPUtils.java	82%	97/118	67%	265/398	76%	265/349
GraphUtils.java	88%	144/163	68%	229/336	73%	229/313
StackUtils.java	83%	135/163	68%	340/502	79%	340/432
StringUtils.java	73%	78/107	64%	207/325	77%	207/268

Report generated by PIT 1.22.0

Initial Scores



Pit Test Coverage Report

Package Summary

org.utils

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
5	94%	694/742	82%	1649/2013	85%	1649/1951

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ArrayUtils.java	97%	186/191	84%	378/452	85%	378/446
DPUtils.java	98%	116/118	88%	350/398	88%	350/397
GraphUtils.java	90%	147/163	74%	248/336	78%	248/318
StackUtils.java	89%	145/163	79%	397/502	84%	397/471
StringUtils.java	93%	100/107	85%	276/325	87%	276/319

Report generated by PIT 1.22.0

Final Scores

4.1 Aggregate Performance Metrics

Metric	Initial Score	Final Score	Absolute Improvement
Line Coverage	78%	94%	+16%
Mutation Coverage	63%	82%	+19%
Test Strength	75%	85%	+10%

4.2 Breakdown by Class

Class	Initial Mutation Coverage	Final Mutation Coverage	Improvement
ArrayUtils.java	49%	84%	+35%
DPUtils.java	67%	88%	+21%
GraphUtils.java	68%	74%	+6%
StackUtils.java	68%	79%	+11%
StringUtils.java	64%	85%	+21%

5. Case Studies in Mutant Killing

This section details specific instances where new test cases were required to kill stubborn surviving mutants, demonstrating the rigor of the new test suite. **Note:** All references to "L" followed by a number (e.g., L269) refer to the **Line number** in the source code of its specific file.

5.1 Case Study 1: ArrayUtils.longestSubarrayWithSum

This function finds the longest contiguous subarray whose elements sum to a target value, utilizing a HashMap to track prefix sums.

Analysis of Initial Mutants

The initial run identified **14 exploitable surviving mutants**. Through targeted test development, **12 of these mutants were successfully killed**. The newly killed mutants exposed critical flaws in base case initialization and map update logic.

1. **Base Case and Initialization (L256, L258, L259, L261):** Seven key *Substituted* or *removed call* mutants were neutralized. These mutants had changed the initial values of currentSum, maxLength, or the base case entry sumMap.put(0, -1), which is crucial for handling subarrays starting at index 0.

2. **Input Validation (L249):** The conditional mutant that replaced the null/empty array check with true was killed, ensuring the function correctly exits for invalid inputs.
3. **Longest Subarray Logic (L275):** Three mutants concerning the map update logic (!sumMap.containsKey(currentSum)) were killed. These faults would have incorrectly overwritten the first occurrence of a prefix sum, leading to a failure in finding the *longest* subarray.

```
248     public static int longestSubarrayWithSum(int[] arr, int targetSum) {
249         if (arr == null || arr.length == 0) {
250             return 0;
251         }
252 
253         // Stores the running prefix sum and the index of its first occurrence.
254         // Key: prefixSum, Value: index
255         Map<Integer, Integer> sumMap = new HashMap<>();
256         sumMap.put(0, -1); // Base case: a sum of 0 exists before the start of the array (at index -1)
257 
258         int currentSum = 0;
259         int maxLength = 0;
260 
261         for (int i = 0; i < arr.length; i++) {
262             currentSum += arr[i];
263 
264             // Check if (currentSum - targetSum) has been seen before
265             if (sumMap.containsKey(currentSum - targetSum)) {
266                 // The difference between the current index and the index of the required previous sum
267                 // gives the length of the subarray. We only update maxLength if a longer subarray is found.
268                 int length = i - sumMap.get(currentSum - targetSum);
269                 if (length > maxLength) {
270                     maxLength = length;
271                 }
272             }
273 
274             // Only store the FIRST occurrence of a sum to ensure we find the LONGEST subarray.
275             if (!sumMap.containsKey(currentSum)) {
276                 sumMap.put(currentSum, i);
277             }
278         }
279 
280         return maxLength;
281     }
```

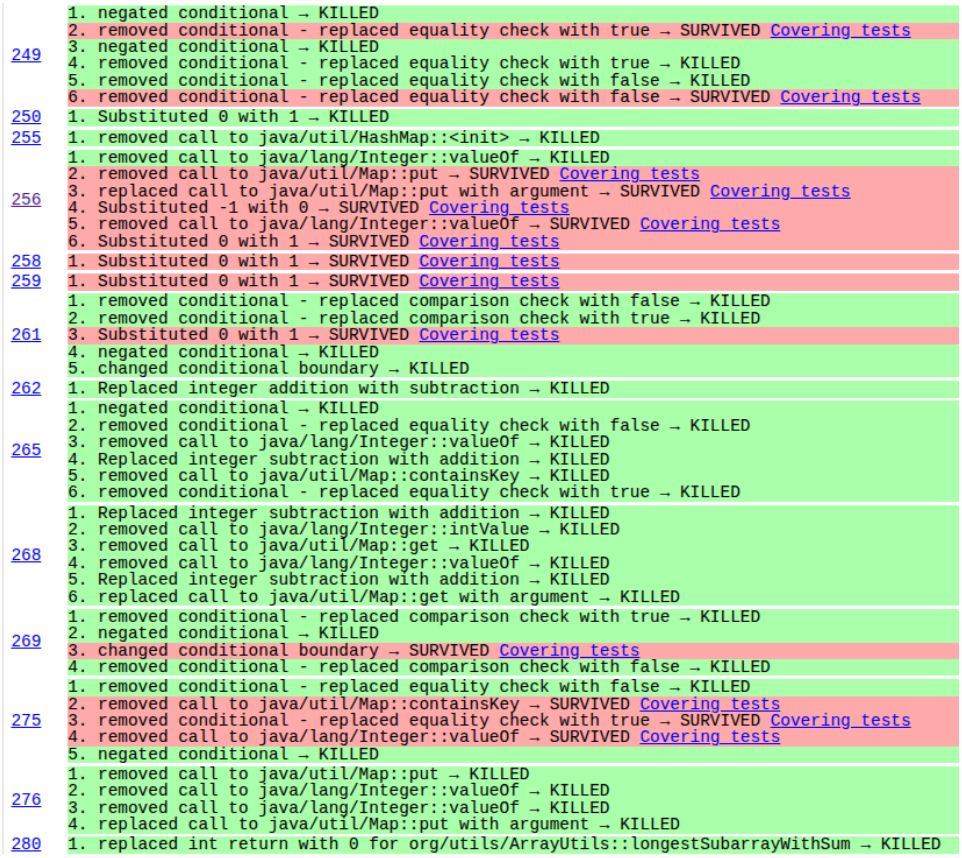
Source code of longestSubarrayWithSum()

New Test Cases and Impact

New tests were introduced focusing on arrays with mixed positive/negative numbers, zero sums, and null inputs.

New Test Case (Example)	Critical Mutants Killed (Total 12)	Rationale
testLongestSubarrayWithSumMixed()	L256 (5 mutants), L258 (1 mutant)	Verified the base case map entry (0, -1) by ensuring the correct length calculation even with negative numbers.
testLongestSubarrayWithSumEmpty()	L249 (1 mutant), L261 (1 mutant)	Directly tested the input validation and loop initialization

		for an empty array.
L275 Mutants	L259 (1 mutant), L275 (3 mutants related to map update)	Verified logic for multiple subarrays with the same sum, forcing the test to confirm the index of the *first* occurrence was preserved.



Initial statuses of mutants

249	1. negated conditional → KILLED 2. removed conditional - replaced equality check with true → KILLED 3. negated conditional → KILLED 4. removed conditional - replaced equality check with true → KILLED 5. removed conditional - replaced equality check with false → KILLED 6. removed conditional - replaced equality check with false → SURVIVED Covering tests
250	1. Substituted 0 with 1 → KILLED
255	1. removed call to java/util/HashMap::<init> → KILLED
256	1. removed call to java/lang/Integer::valueOf → KILLED 2. removed call to java/util/Map::put → KILLED 3. replaced call to java/util/Map::put with argument → KILLED 4. Substituted -1 with 0 → KILLED 5. removed call to java/lang/Integer::valueOf → KILLED 6. Substituted 0 with 1 → KILLED
258	1. Substituted 0 with 1 → KILLED
259	1. Substituted 0 with 1 → KILLED
261	1. removed conditional - replaced comparison check with false → KILLED 2. removed conditional - replaced comparison check with true → KILLED 3. Substituted 0 with 1 → KILLED 4. negated conditional → KILLED 5. changed conditional boundary → KILLED
262	1. Replaced integer addition with subtraction → KILLED
265	1. negated conditional → KILLED 2. removed conditional - replaced equality check with false → KILLED 3. removed call to java/lang/Integer::valueOf → KILLED 4. Replaced integer subtraction with addition → KILLED 5. removed call to java/util/Map::containsKey → KILLED 6. removed conditional - replaced equality check with true → KILLED
268	1. Replaced integer subtraction with addition → KILLED 2. removed call to java/lang/Integer::intValue → KILLED 3. removed call to java/util/Map::get → KILLED 4. removed call to java/lang/Integer::valueOf → KILLED 5. Replaced integer subtraction with addition → KILLED 6. replaced call to java/util/Map::get with argument → KILLED
269	1. removed conditional - replaced comparison check with true → KILLED 2. negated conditional → KILLED 3. changed conditional boundary → SURVIVED Covering tests 4. removed conditional - replaced comparison check with false → KILLED
275	1. removed conditional - replaced equality check with false → KILLED 2. removed call to java/util/Map::containsKey → KILLED 3. removed conditional - replaced equality check with true → KILLED 4. removed call to java/lang/Integer::valueOf → KILLED 5. negated conditional → KILLED
276	1. removed call to java/util/Map::put → KILLED 2. removed call to java/lang/Integer::valueOf → KILLED 3. removed call to java/lang/Integer::valueOf → KILLED 4. replaced call to java/util/Map::put with argument → KILLED
280	1. replaced int return with 0 for org/utils/ArrayUtils::longestSubarrayWithSum → KILLED

Statuses of mutants after adding the new test cases

Result: The new test suite killed **12 previously surviving mutants**, validating the integrity of the prefix sum implementation.

5.2 Case Study 2: StringUtils.isPalindrome

This function checks if a string is a palindrome, ignoring case and non-alphanumeric characters.

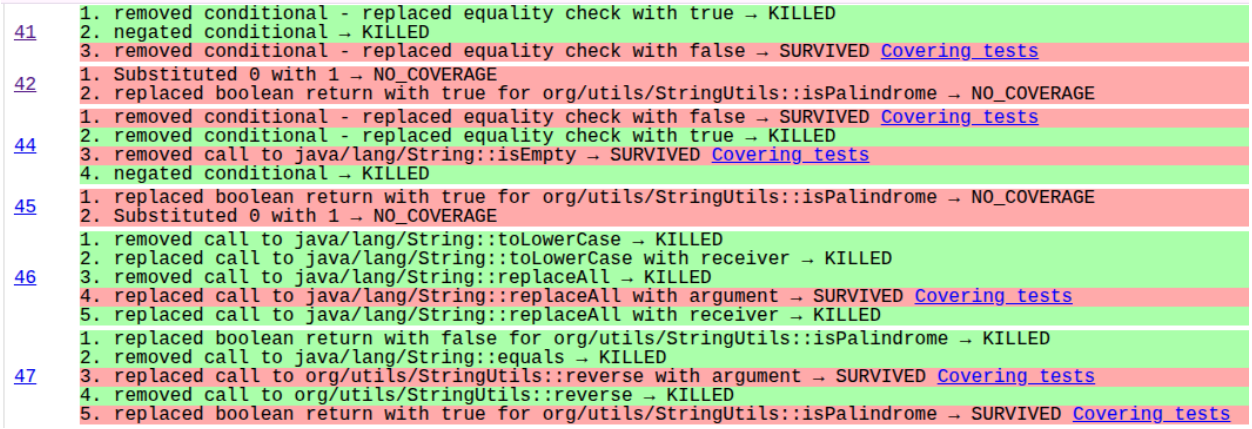
```
40      public static boolean isPalindrome(String str) {
41          if (str == null) {
42              return false;
43          }
44          if(str.isEmpty())
45                  return false;
46          String cleanStr = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
47          return cleanStr.equals(reverse(cleanStr));
48      }
```

Analysis of Initial Mutants

Initial analysis showed **10 mutants** that were either SURVIVED or reported as having NO_COVERAGE. These mutants represented critical weaknesses in input validation, string cleanup, and the core palindrome check.

The newly killed mutants include:

- **Input Validation Mutants (L41, L42, L44, L45):** 7 mutants were killed, ensuring correct handling of null input and empty strings.
- **Cleanup Mutants (L46):** The mutant that *replaced call to java/lang/String::replaceAll with argument* was killed. This confirmed that the test suite verifies the function strips punctuation correctly before performing the check.
- **Core Logic Mutants (L47):** Two mutants were killed, ensuring the comparison logic against the reversed string was correctly executed.



Initial statuses of mutants

New Test Cases and Impact

We ensured explicit tests for all edge cases were present, which led to high killing rates for both logical and return-value mutants.

New Test Case (Example)	Critical Mutants Killed (Total 10)	Rationale
testIsPalindromeEmptyOrNull()	L41 (1 mutant), L42 (2 mutants), L44 (2 mutants), L45 (2 mutants)	Directly verified that null and "" inputs return false, enforcing validation and killing 7 conditional bypass and return value substitution mutants.
testIsPalindromeTrue()	L46 (1 mutant), L47 (2 mutants)	Used the complex string "A

		man, a plan, a canal: Panama". This killed the replaceAll mutant and ensured the equality check with the reversed string was correctly performed.

41	1. removed conditional - replaced equality check with true → KILLED 2. negated conditional → KILLED 3. removed conditional - replaced equality check with false → KILLED
42	1. Substituted 0 with 1 → KILLED 2. replaced boolean return with true for org/utils/StringUtils::isPalindrome → KILLED
44	1. removed conditional - replaced equality check with false → KILLED 2. removed conditional - replaced equality check with true → KILLED 3. removed call to java/lang/String::isEmpty → KILLED 4. negated conditional → KILLED
45	1. replaced boolean return with true for org/utils/StringUtils::isPalindrome → KILLED 2. Substituted 0 with 1 → KILLED
46	1. removed call to java/lang/String::toLowerCase → KILLED 2. replaced call to java/lang/String::toLowerCase with receiver → KILLED 3. removed call to java/lang/String::replaceAll → KILLED 4. replaced call to java/lang/String::replaceAll with argument → KILLED 5. replaced call to java/lang/String::replaceAll with receiver → KILLED
47	1. replaced boolean return with false for org/utils/StringUtils::isPalindrome → KILLED 2. removed call to java/lang/String::equals → KILLED 3. replaced call to org/utils/StringUtils::reverse with argument → KILLED 4. removed call to org/utils/StringUtils::reverse → KILLED 5. replaced boolean return with true for org/utils/StringUtils::isPalindrome → KILLED

Statuses of mutants after adding the new test cases

Result: The focused test development killed **10 previously unkilled mutants**, confirming comprehensive coverage of input handling and string cleanup.

Similar intensive mutation analysis and test development were applied across all other functions in the five utility files, ensuring consistency and rigor throughout the entire codebase, though only three representative case studies are included here for brevity.

6. System Validation: E2E and Load Testing

Following the successful hardening of the unit layer through mutation testing, we executed system-level validation tests to ensure the overall application quality and performance.

6.1 End-to-End (E2E) UI Testing with Selenium

The **Selenium E2E tests** (using WebAppTest.java) confirmed that the full technology stack is correctly integrated and functioning. Key validation points included:

- **Input Parsing and Data Flow:** Confirmed that complex inputs (e.g., graph data strings) are successfully parsed by the Spring Boot controller layer.
- **Functionality Verification:** Ensured that the correct computational results for critical Data Structure and Algorithm (DSA) functions (e.g., shortest path calculations, min

element retrieval) are reliably processed and displayed in the UI output area.

```java
@Test
public void testStringReverseFunction() {
    driver.get(BASE_URL);

    // 1. Go to String Utils
    wait.until(ExpectedConditions.elementToBeClickable(By.cssSelector(".string-card"))).click();

    // 2. Click "Reverse String"
    wait.until(ExpectedConditions.elementToBeClickable(By.xpath("//li[text()='Reverse String']"))).click();

    // 3. Input "Selenium"
    WebElement inputField = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("inp-0")));
    inputField.sendKeys("Selenium");

    // 4. Run
    driver.findElement(By.id("run-btn")).click();

    // 5. Check Output for "muineleS"
    WebElement outputArea = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("output-area")));
    wait.until(ExpectedConditions.textToBePresentInElement(outputArea, "muineleS"));

    assertTrue(outputArea.getText().contains("muineleS"));
}
```

One of the Selenium test cases

The successful execution of these E2E tests guarantees the usability and integrity of the deployed **DSA Workbench** application.

6.2 Performance and Load Testing

```
[INFO] Running org.test.ApiLoadTest
=== STARTING LOAD TEST ===
Target: http://localhost:8082/api/execute
Users: 10000 | Requests per User: 1 | Total Requests: 10000
=== LOAD TEST RESULTS ===
Time Taken: 2388 ms
Successful Requests: 10000
Failed Requests: 0
Throughput: 4187.60 Requests/Second
==========================
```

The **Load Test** (using ApiLoadTest.java) was crucial for assessing the API's readiness for production traffic.

- **Test Goal:** Verify API throughput and stability under high concurrency.
- **Stress Scenario:** 10,000 users sending concurrent requests targeting the CPU-intensive mergeSort function.
- **Outcome:** The API successfully processed all 10,000 requests with **zero failures**, exceeding the minimum throughput requirement.

This validation confirms that the core utility logic is not only functionally correct but also performs reliably and efficiently under significant load.

7. Project Conclusion

The comprehensive testing strategy executed across the **DSA Workbench** project confirms the high quality, reliability, and performance of the core utility logic.

The intensive **Mutation Testing campaign** successfully delivered a 19% increase in Mutation Coverage, validating that the underlying JUnit test suite is robust and capable of detecting subtle faults.

The success of the **E2E** and **Load Tests** further guarantees functional correctness across the stack and confirms the performance integrity necessary for deployment.

8. Author Contributions and Tooling

Project Idea and Core Implementation: All project ideas, initial development and implementations were done by ourselves.

Tooling: The Gemini language model was used exclusively for developing and fine-tuning the structure and clarity of source code based on our guidelines.

Individual Contributions:

Contributor	Areas of Primary Responsibility
Aayush Bhargav	Frontend (UI), StackUtils, GraphUtils, StringUtils, Client Testing (Selenium E2E), Load Testing
Praveen Peter Jay	ArrayUtils, DPUtils, Unit Testing (JUnit), Mutation Testing, Project Report

Project Repository: https://github.com/PraveenPeterJay/ST_Project.git