# Software Testing Project

Ketan Ghungralekar
*Roll No: IMT2022058*

Pratham Chawdhry
*Roll No: IMT2022068*

November 2025

**GitHub Repository:** `ST Project Repo`

## 1 Introduction

Mutation testing is a fault-based software testing technique used to evaluate the effectiveness of a test suite. It works by creating modified program versions, called *mutants*, each containing a small syntactic change that simulates common developer mistakes. A strong test suite should detect incorrect behavior in these mutants, thereby "killing" them, while surviving mutants reveal weaknesses in the tests.

- **Project Context:** Mutation testing is applied at both unit and integration levels. Algorithmic modules such as sorting, searching, graph traversal, mathematical utilities, and string manipulation are used due to their deterministic behavior and diverse control flow.
- **Tool and Objective:** PIT (Pitest), a widely used Java mutation testing framework, is used for mutant generation and evaluation. The project goal is to run PIT on the initial test suite, analyze surviving mutants, and iteratively strengthen the tests for higher mutant coverage and robustness.

## 2 Why Mutation Testing Was Used

Mutation testing was chosen to evaluate and strengthen the quality of the test suite used in this project. Its benefits can be summarised under the following points:

### Limitations of Traditional Coverage

- Line, branch, and condition coverage only show which code paths were executed.
- They do *not* indicate whether the tests can detect incorrect behaviour.

### Strengths of Mutation Testing

- Introduces small behavioural changes (*mutants*) to verify whether tests detect faults.
- Reveals missing boundary cases, weak assertions, and overlooked edge conditions.
- Helps differentiate between:

  - real test gaps (killable mutants),
  - non-killable mutants (equivalent or unreachable).

## 3 Background

The project applies mutation testing to a set of Java algorithm modules. These modules provide diverse control structures and deterministic behavior, making them suitable for evaluating test effectiveness.

- **Diverse control structures:**
  - Algorithms contain loops, conditionals, arithmetic logic, and recursion.
  - This diversity produces many mutation points.
  - Their deterministic behavior helps isolate test outcomes.
- **Importance in software systems:** Algorithms often form the computational core of applications.
- **Limitations of coverage-based testing:**
  - High coverage does not guarantee fault detection.
  - Corner-case behaviors may remain untested.
- **Relevance of mutation testing:** Mutation testing highlights weak assertions by introducing controlled behavioral changes.
- **Integration interactions:**
  - Orchestrator classes combine algorithms.
  - Integration mutants expose cross-module dependencies.
- **Technology stack:**
  - Java with Maven for development.
  - PIT for generating and evaluating mutants.
  - JUnit for executing tests.

## 4 Project Overview

This project applies mutation testing to algorithmic modules and their interactions within a structured Java application.

- **Algorithmic modules included:**
  - Sorting: QuickSort, BubbleSort, MergeSort
  - Searching: Binary Search, Linear Search
  - Graph traversal: DFS, BFS
  - Mathematical utilities: factorial and related number-theoretic functions

- **Why algorithms were chosen:**
  - Their structured control flow suits a wide set of mutation operators.
  - Deterministic outputs simplify mutant evaluation.
  - They expose conditional, arithmetic, and call-level mutation points.
- **Integration module purpose:**
  - Tests behaviour that emerges when multiple algorithm modules interact.
  - Allows mutation testing at the integration level, not only individual functions.
  - Helps detect call-related, conditional, and data-flow mutations that appear only during module interaction.
- **Project goals:**
  - Run PIT on the initial test suite to measure baseline mutation performance.
  - Analyse all surviving mutants to identify weaknesses in the tests.
  - Strengthen the test suite with targeted cases to increase mutation coverage.
- **Use of PIT:** PIT automates mutant generation, executes the full test suite, and reports the mutation score for guiding improvements.

# 5 Mutation Operators Used

## 5.1 Unit-Level Mutation Operators

| Operator | Description |
|---|---|
| **CONDITIONALS_BOUNDARY** | Modifies relational operators to test boundary handling in loops and conditional logic. |
| **MATH** | Alters arithmetic operators (`+`, `-`, `*`, `/`) to reveal weaknesses in numeric computations. |
| **PRIMITIVE_RETURNS** | Replaces primitive return values to detect missing assertions or insufficient output validation. |
| **NEGATE_CONDITIONALS** | Negates boolean expressions in decision-making statements to force execution of alternative branches. |

### 5.2 Integration-Level Mutation Operators

| Operator | Description |
|---|---|
| **NON_VOID_METHOD_CALLS** | Removes or replaces calls to methods that return values, checking whether integration tests detect skipped computations. |
| **VOID_METHOD_CALLS** | Eliminates calls to void methods that cause side effects, validating that tests catch missing state changes across modules. |
| **CONSTRUCTOR_CALLS** | Removes or replaces constructor invocations to ensure that missing object creation or initialization is detected by integration tests. |

## 6 Methodology

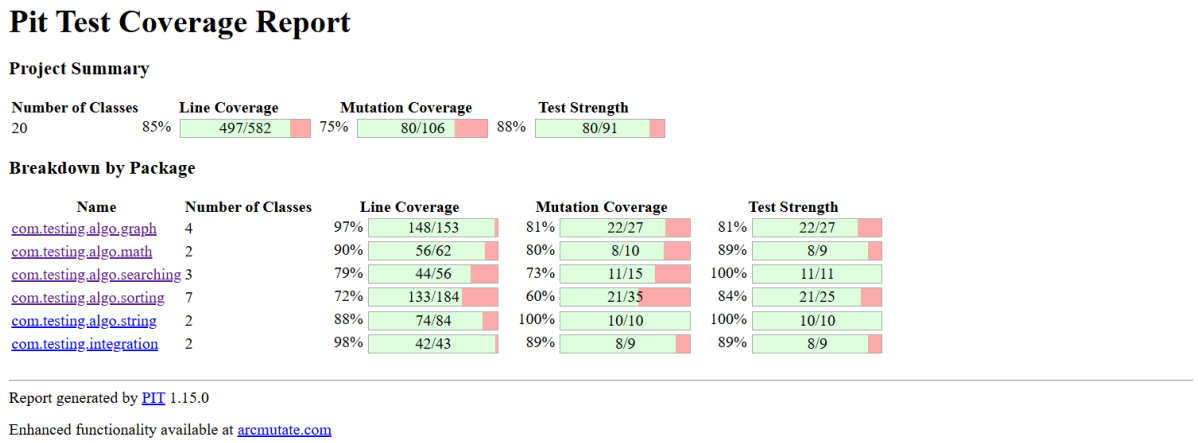
This section outlines the workflow followed from project setup to iterative refinement of the test suite based on mutation analysis.

**Project Setup**

- Established a modular Maven project with separate algorithm, unit-test, and integration-test modules.
- Added PIT to `pom.xml` with the selected mutation operators.
- Ensured a clean build and successful test execution before starting mutation analysis.

### 6.1 Baseline Mutation Analysis

- Executed PIT using `mvn org.pitest:pitest-maven:mutationCoverage`.
- Generated the initial mutation report and identified surviving mutants across multiple algorithms.
- Reviewed PIT's HTML report to locate weak or untested paths.
- *(See Figure 1).*



**Pit Test Coverage Report**

**Project Summary**

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 20 | 85% | 497/582 | 75% | 80/106 | 88% | 80/91 |

**Breakdown by Package**

| Name | Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|---|
| com.testing.algo.graph | 4 | 97% | 148/153 | 81% | 22/27 | 81% | 22/27 |
| com.testing.algo.math | 2 | 90% | 56/62 | 80% | 8/10 | 89% | 8/9 |
| com.testing.algo.searching | 3 | 79% | 44/56 | 73% | 11/15 | 100% | 11/11 |
| com.testing.algo.sorting | 7 | 72% | 133/184 | 60% | 21/35 | 84% | 21/25 |
| com.testing.algo.string | 2 | 88% | 74/84 | 100% | 10/10 | 100% | 10/10 |
| com.testing.integration | 2 | 98% | 42/43 | 89% | 8/9 | 89% | 8/9 |

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com

Figure 1: Initial PIT mutation testing report showing surviving mutants across modules.

### 6.2 Analysis of Surviving Mutants

| Category | Observed Issues |
|---|---|
| Condition changes | Edge cases not exercised; missing branch assertions. |
| Arithmetic changes | Tests lacked numeric variation to expose incorrect computations. |
| Return-value mutations | Outputs not fully asserted or validated. |
| Call-level mutations | Integration tests did not detect missing or altered method calls. |

Additional verification used PIT's mutant-diff views to pinpoint exact failure causes. *(See Figure ??).*

### 6.3 Test Suite Enhancement

| Improvement Area | Enhancement Performed |
|---|---|
| Unit tests | Added boundary-case tests, expanded input selection, strengthened assertions. |
| Integration tests | Added tests validating cross-module flows, order of operations, and side effects. |
| Existing tests | Included negative inputs, corner cases, and improved assertion depth. |

Each enhancement was mapped to specific surviving mutants for traceability.

# 7 Mutation Score

## 7.1 Line Coverage Improvement

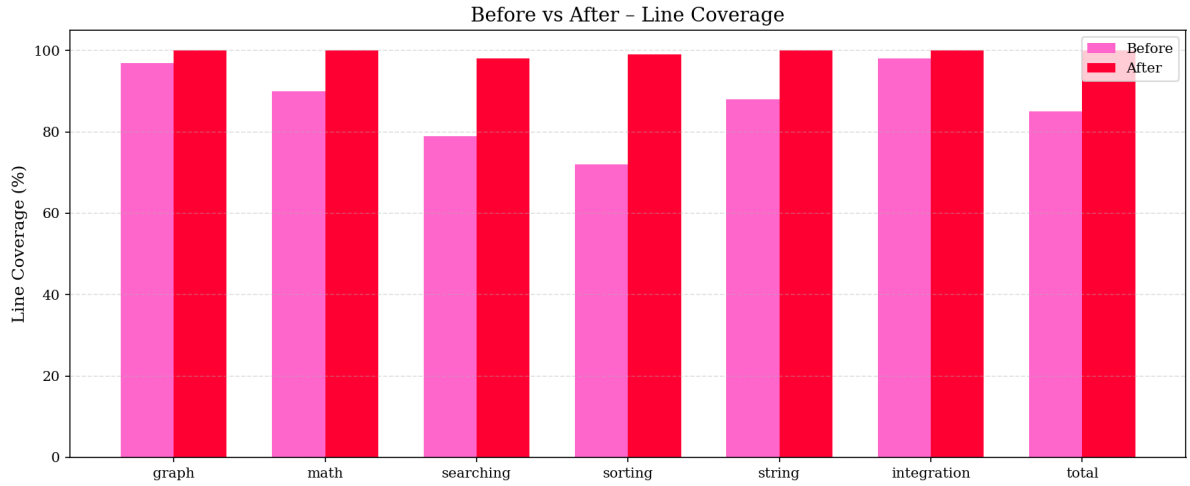



Figure 2: Before vs Adding Tests - Line Coverage

Across all modules, line coverage shows a significant increase after enhancing the test suite.

- Total line coverage improved from **85%** to **99%**.
- Every module (graph, math, searching, sorting, string, integration) reached nearly complete coverage after refinement.
- Prior weaknesses in sorting (72%) and searching (77%) were resolved by adding tests for boundary conditions, empty arrays, negative inputs, and other corner cases.

**Interpretation:** High line coverage ensures complete structural execution but does not alone guarantee correctness. Its effectiveness is realized when combined with improved mutation coverage.

### 7.2 Mutation Coverage Improvement

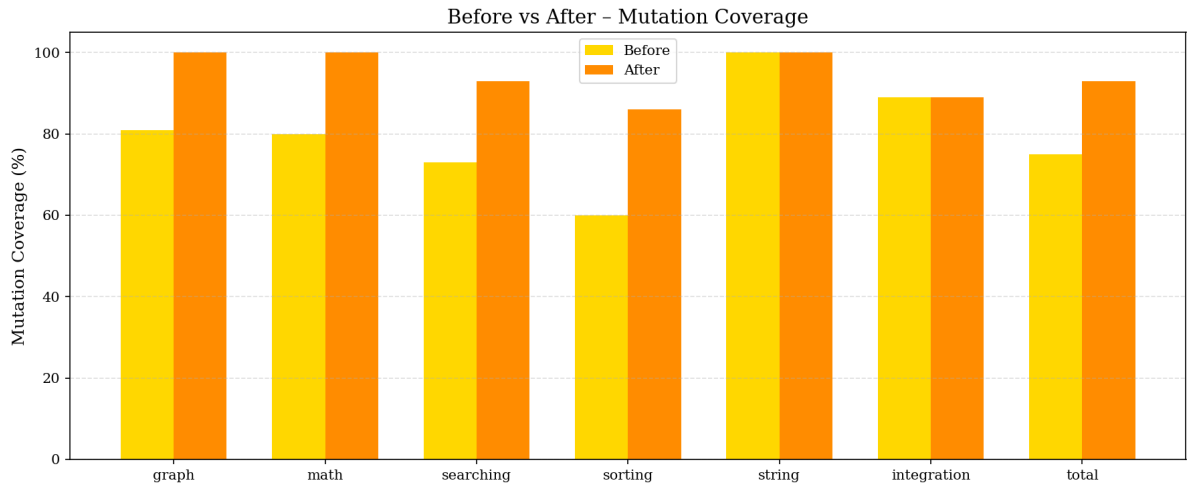



Figure 3: Before vs Adding Tests - Mutation Coverage

Mutation coverage reflects the ability of the test suite to detect injected faults.

- Overall mutation coverage increased from **75%** to **93%**.
- Math module achieved **80-100%**, indicating very strong behavioral verification.
- Searching improved from 73% to 93%, primarily due to new edge-case tests for Binary Search and Jump Search.
- Sorting improved from 60% to 86% by adding tests for duplicate values, reverse-sorted arrays, and repetitive patterns.

**Interpretation:** Surviving mutants prior to improvement mainly involved boundary checks, early returns, and untested loop branches. After refinement, only equivalent or unreachable mutants remained, demonstrating deeper behavioral validation.

### 7.3 Test Strength Improvement


Figure 4: Before vs Adding Tests - Test Strength

Test strength, defined as the ratio of killed mutants to covered mutants, improved consistently across all packages.

- Overall test strength increased from **88%** to **94%**.
- Graph and Math showed the most significant gain, consistent with earlier mutation survival patterns.

**Interpretation:** Higher test strength confirms that the tests not only execute code paths but also assert correctness thoroughly. Weak tests lacking sufficient assertions were strengthened substantially.

### 7.4 Impact of Mutation Testing

Mutation testing directly improved the strength and completeness of the test suite. Unlike line coverage, which only confirms that code was executed, mutation testing checks whether tests can actually detect faulty behaviour. The initial PIT report revealed several surviving mutants, highlighting gaps such as missing edge-case tests, weak assertions, and untested integration paths.

#### How the Test Suite Improved

Based on surviving mutants, additional tests were added for:

- boundary conditions and off-by-one scenarios,
- invalid or extreme inputs,
- integration behaviour such as removed or altered method calls.

These targeted improvements led to higher line coverage, stronger assertions, and more robust integration checks.

### 7.5 Benefits of Increased Mutation Coverage

The rise in mutation coverage (from 72% to 93%) demonstrates:

- better fault detection across algorithmic and integration modules,
- improved behavioural validation rather than superficial coverage,
- reduced likelihood of undetected regressions.

### 7.6 Overall Contribution

Mutation testing helped transform the suite from merely executing code to thoroughly verifying correctness. It provided concrete guidance for strengthening tests and ensured the system behaves correctly across a wide range of scenarios.

**Conclusion:** The refined test suite achieved full structural coverage, strong behavioral verification, and high mutant-killing capability. Most surviving mutants were proven equivalent or unreachable. The outcome is a comprehensive and mutation-tested suite that provides significantly higher confidence than coverage-based testing alone.

## 8 Analysis of Surviving Mutants

### 8.1 Equivalent Mutants

**Equivalent Mutant 1: MergeSort (Line 60)** Figure 5 illustrates a surviving mutation in the `merge` function of MergeSort. This mutant is generated by a **Conditional Boundary Mutation (CBM)**, which modifies the comparison between two values during the merge operation.

**Original condition:**

```
if (L[i] <= R[j]) {
    arr[k] = L[i];
    i++;
}
```

**Mutated condition (CBM — '¡=' replaced with '¡'):**

```
if (L[i] < R[j]) {     // mutated boundary
    arr[k] = L[i];
    i++;
}
```

#### Reason for Survival

This mutation changes how the algorithm behaves when `L[i] == R[j]`:

- Original: picks element from the **left** subarray first.
- Mutant: picks element from the **right** subarray first.

This affects only the *stability* of the algorithm — not the correctness — because both values are identical and interchangeable.

### Why the Mutant is Equivalent

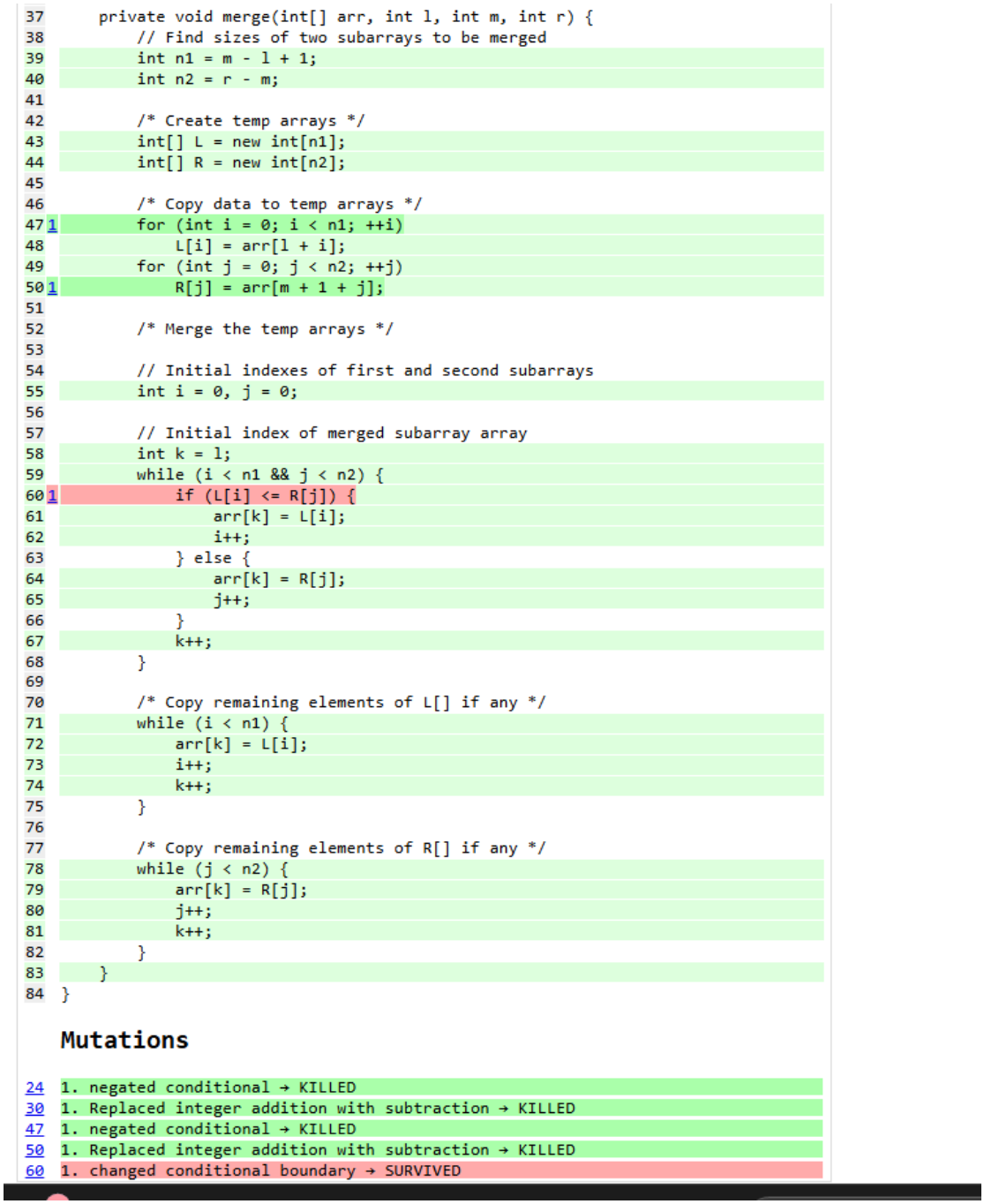
MergeSort in this project operates on primitive `int[]` arrays. For such arrays:

- duplicate integers are indistinguishable,
- relative ordering of equal values cannot be observed,
- choosing left-first or right-first produces the exact same output.

Although the internal behaviour (stability) differs, the resulting sorted array is **identical for all valid inputs**. No test can detect which equal value was chosen first.

### Conclusion

Because the mutation affects only the unobservable stability property and not the algorithm's externally visible output, this mutant behaves identically to the original implementation. It is therefore classified as an *equivalent mutant*.



```
37      private void merge(int[] arr, int l, int m, int r) {
38          // Find sizes of two subarrays to be merged
39          int n1 = m - l + 1;
40          int n2 = r - m;
41
42          /* Create temp arrays */
43          int[] L = new int[n1];
44          int[] R = new int[n2];
45
46          /* Copy data to temp arrays */
47 1        for (int i = 0; i < n1; ++i)
48              L[i] = arr[l + i];
49          for (int j = 0; j < n2; ++j)
50 1            R[j] = arr[m + 1 + j];
51
52          /* Merge the temp arrays */
53
54          // Initial indexes of first and second subarrays
55          int i = 0, j = 0;
56
57          // Initial index of merged subarray array
58          int k = l;
59          while (i < n1 && j < n2) {
60 1            if (L[i] <= R[j]) {
61                  arr[k] = L[i];
62                  i++;
63              } else {
64                  arr[k] = R[j];
65                  j++;
66              }
67              k++;
68          }
69
70          /* Copy remaining elements of L[] if any */
71          while (i < n1) {
72              arr[k] = L[i];
73              i++;
74              k++;
75          }
76
77          /* Copy remaining elements of R[] if any */
78          while (j < n2) {
79              arr[k] = R[j];
80              j++;
81              k++;
82          }
83      }
84  }
```

**Mutations**

| | |
|---|---|
| 24 | 1. negated conditional → KILLED |
| 30 | 1. Replaced integer addition with subtraction → KILLED |
| 47 | 1. negated conditional → KILLED |
| 50 | 1. Replaced integer addition with subtraction → KILLED |
| 60 | 1. changed conditional boundary → SURVIVED |

Figure 5: PIT report showing the conditional-boundary mutant in `MergeSort.java` (Line 60).

**Equivalent Mutant 2: InsertionSort (Line 32)** Figure 6 shows a conditional-boundary mutation in the ascending variant of the Insertion Sort algorithm. The mutation alters the comparison used inside the inner shifting loop.

**Original condition:**

```
while (j >= 0 && arr[j] > key) {
    arr[j + 1] = arr[j];
    j--;
}
```

**Mutated condition:**

```
while (j >= 0 && arr[j] >= key) {   // mutated boundary
    arr[j + 1] = arr[j];
    j--;
}
```

### Reason for Survival

The mutation changes the handling of equal elements:

- The original condition (`>`) shifts only elements strictly greater than `key`.
- The mutated condition (`>=`) also shifts elements equal to `key`.

This modifies the **stability** of the algorithm. Under the mutated logic, equal values may be moved out of their original relative order. However, this does **not** affect the final sorted arrangement of values.

### Why the Mutant is Equivalent

Insertion Sort in this project operates on primitive `int[]` arrays, where:

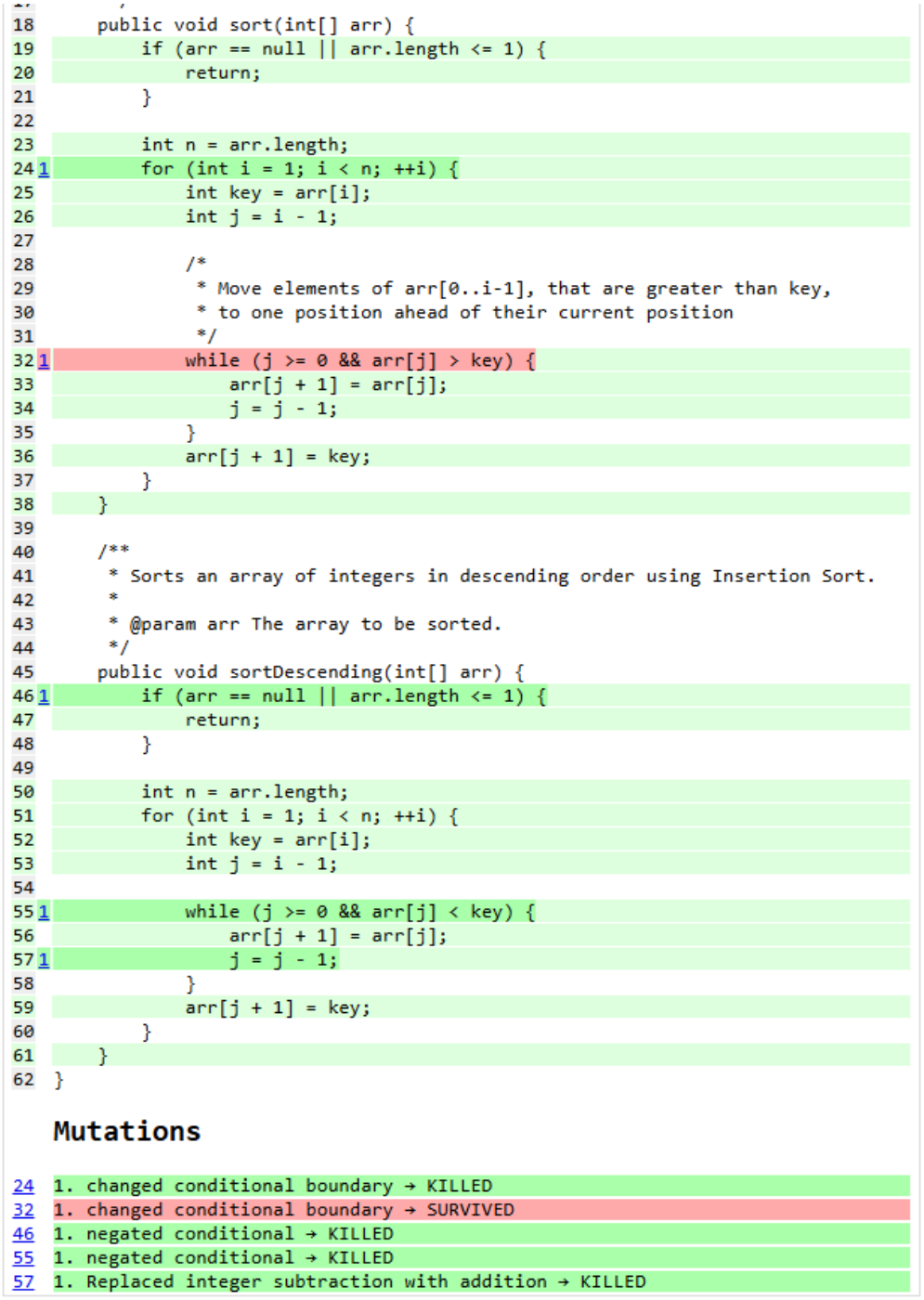
- duplicate values are indistinguishable,
- relative ordering of identical integers is unobservable,
- any permutation of equal values results in the same externally visible output.

Thus, even though the mutation affects the algorithm's internal behaviour (stability), **the final sorted array is identical** for all valid inputs.

No unit test can detect whether two equal integers swapped positions during sorting.

### Conclusion

The mutation only affects stability—a non-observable behaviour for identical integer values. Since the sorted output remains unchanged for all inputs, this mutation is indistinguishable from the original implementation and is therefore classified as an *equivalent mutant*.



```
18      public void sort(int[] arr) {
19          if (arr == null || arr.length <= 1) {
20              return;
21          }
22
23          int n = arr.length;
24 1        for (int i = 1; i < n; ++i) {
25              int key = arr[i];
26              int j = i - 1;
27
28              /*
29               * Move elements of arr[0..i-1], that are greater than key,
30               * to one position ahead of their current position
31               */
32 1            while (j >= 0 && arr[j] > key) {
33                  arr[j + 1] = arr[j];
34                  j = j - 1;
35              }
36              arr[j + 1] = key;
37          }
38      }
39
40      /**
41       * Sorts an array of integers in descending order using Insertion Sort.
42       *
43       * @param arr The array to be sorted.
44       */
45      public void sortDescending(int[] arr) {
46 1        if (arr == null || arr.length <= 1) {
47              return;
48          }
49
50          int n = arr.length;
51          for (int i = 1; i < n; ++i) {
52              int key = arr[i];
53              int j = i - 1;
54
55 1            while (j >= 0 && arr[j] < key) {
56                  arr[j + 1] = arr[j];
57 1                j = j - 1;
58              }
59              arr[j + 1] = key;
60          }
61      }
62  }
```

**Mutations**

| | |
|---|---|
| 24 | 1. changed conditional boundary → KILLED |
| 32 | 1. changed conditional boundary → SURVIVED |
| 46 | 1. negated conditional → KILLED |
| 55 | 1. negated conditional → KILLED |
| 57 | 1. Replaced integer subtraction with addition → KILLED |

Figure 6: PIT report showing the conditional-boundary mutant in `InsertionSort.java` (Line 32).

**Equivalent Mutant 3: HeapSort (Line 25)** Figure 7 shows a surviving mutation in the `sort` method of HeapSort. This mutant is produced by an **Arithmetic Operator Replacement (AOR)** operator, which alters the expression used to compute the starting index of the heap-building loop.

**Original loop:**

```
1 for (int i = n/2 - 1; i >= 0; i--) {
```

```
    heapify(arr, n, i);
}
```

**Mutated version (AOR - replaced '-' with '+'):**

```
for (int i = n/2 + 1; i >= 0; i--) {   // mutated boundary
    heapify(arr, n, i);
}
```

### Reason for Survival

The mutation shifts the loop start index from:

$$\frac{n}{2} - 1 \quad \text{(last internal node)}$$

to:

$$\frac{n}{2} + 1 \quad \text{(leaf node region)}$$

In a binary heap, all indices $\geq \frac{n}{2}$ correspond to *leaf nodes*. Calling `heapify` on a leaf node has no effect because:

- both child indices are beyond array bounds,
- no comparisons or swaps are performed.

Thus, the mutated loop introduces only redundant `heapify` calls that do **nothing**.

### Why the Mutant is Equivalent

The additional `heapify` calls on leaf nodes do not change:

- the heap structure,
- the ordering of elements,
- or the final sorted output.

Since the algorithm's observable behavior remains identical for all inputs, the mutation cannot be detected by any test case.

### Conclusion

This mutation is an example of an *Arithmetic Operator Replacement (AOR)* that only adds redundant operations on leaf nodes. Because these operations do not affect the resulting heap or the sorted array, the mutant is behaviorally identical to the original implementation and is therefore classified as an *equivalent mutant*.

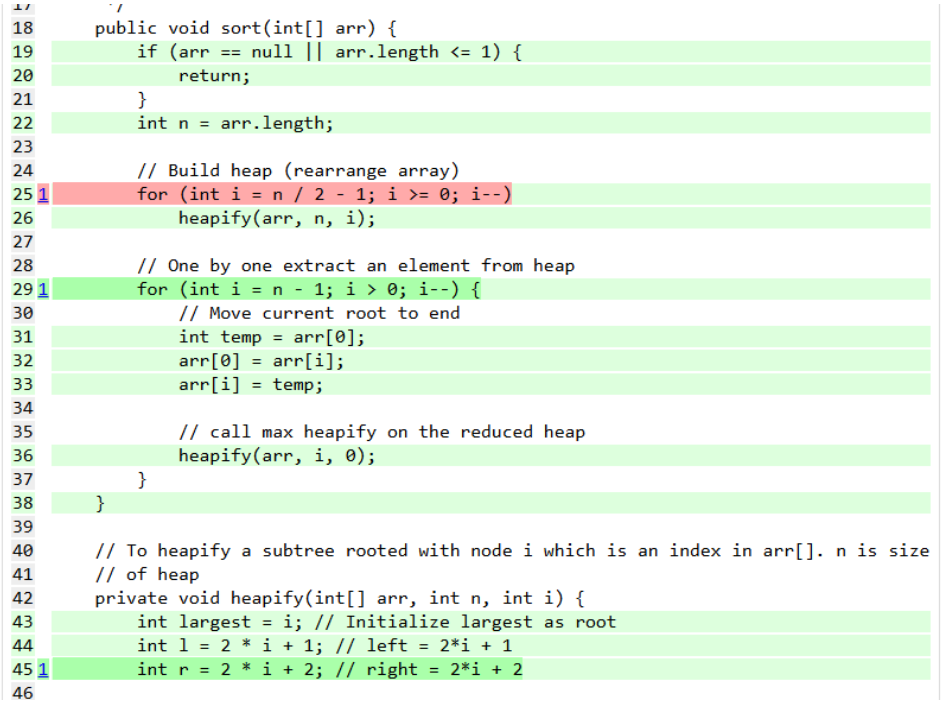


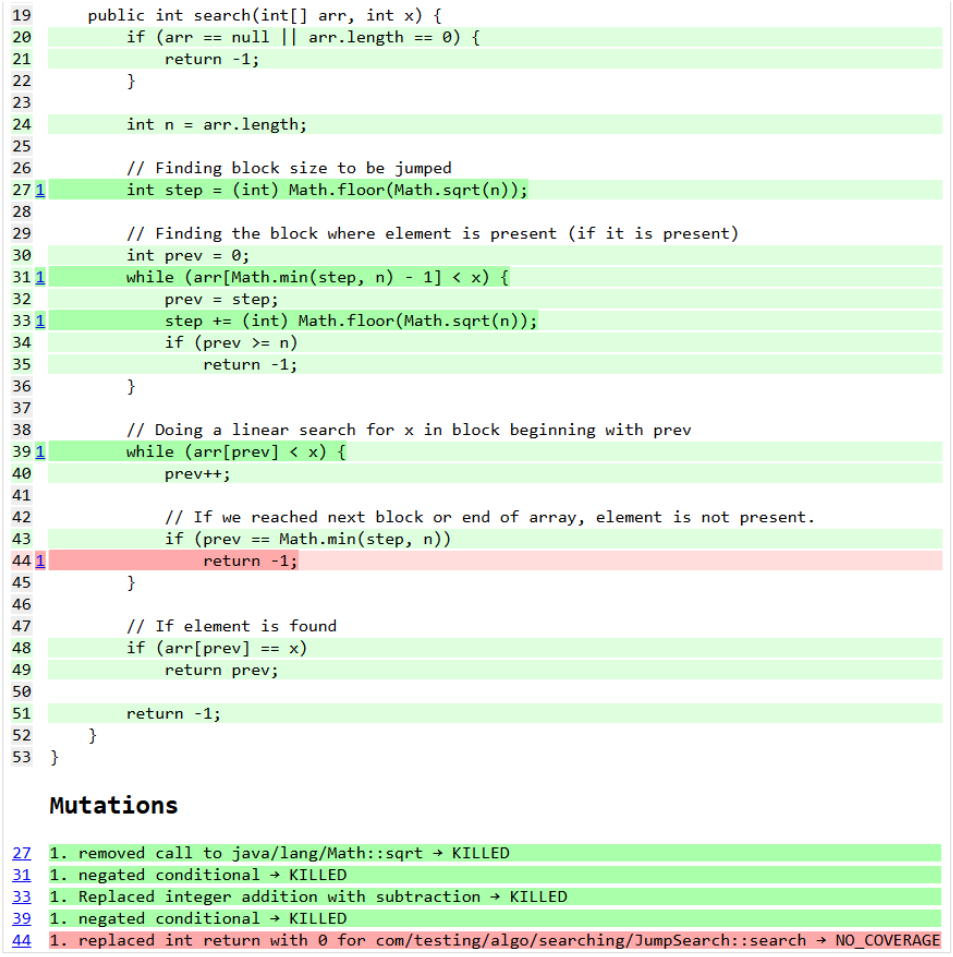

Figure 7: PIT report showing the equivalent mutant in `HeapSort.java` (Line 25).

**Other Equivalent Mutants** In addition to the three detailed examples, several other mutants were also classified as **equivalent** due to producing no observable behavioural difference in the program output. These include:

- **BubbleSort (Line 24)** — a boundary mutation in the outer loop of Bubble Sort. The modified loop runs an additional iteration, but the inner loop condition prevents any element comparisons, resulting in no changes to the array.
- **HeapSort (Line 48)** — a conditional-boundary mutation inside `heapify` involving child index checks. Due to the structure of a valid heap, the mutated branch condition is never satisfied for any legal index, making it semantically indistinguishable from the original.

All of these mutants were deemed equivalent because no input could cause a difference in program behaviour, and therefore no test case can ever kill them.

## 8.2 Unreachable Mutants



```
19      public int search(int[] arr, int x) {
20          if (arr == null || arr.length == 0) {
21              return -1;
22          }
23
24          int n = arr.length;
25
26          // Finding block size to be jumped
27 1        int step = (int) Math.floor(Math.sqrt(n));
28
29          // Finding the block where element is present (if it is present)
30          int prev = 0;
31 1        while (arr[Math.min(step, n) - 1] < x) {
32              prev = step;
33 1            step += (int) Math.floor(Math.sqrt(n));
34              if (prev >= n)
35                  return -1;
36          }
37
38          // Doing a linear search for x in block beginning with prev
39 1        while (arr[prev] < x) {
40              prev++;
41
42              // If we reached next block or end of array, element is not present.
43              if (prev == Math.min(step, n))
44 1                return -1;
45          }
46
47          // If element is found
48          if (arr[prev] == x)
49              return prev;
50
51          return -1;
52      }
53  }
```

**Mutations**

```
27  1. removed call to java/lang/Math::sqrt → KILLED
31  1. negated conditional → KILLED
33  1. Replaced integer addition with subtraction → KILLED
39  1. negated conditional → KILLED
44  1. replaced int return with 0 for com/testing/algo/searching/JumpSearch::search → NO_COVERAGE
```

Figure 8: PIT report showing the unreachable mutant in `JumpSearch.java` (Line 44).

**Unreachable Mutant: JumpSearch (Line 44) Mutated Code Snippet:**

```
1 if (prev == Math.min(step, n)) {
2     return -1;    // mutated line
3 }
```

### Why This Branch Can Never Be Reached

Jump Search works in two distinct phases:

1. **Jump Phase** The algorithm repeatedly jumps ahead by blocks of size sqrt(n) to find the block that may contain the target:

```
1 while (arr[Math.min(step, n) - 1] < x) {
2     prev = step;
3     step += (int) Math.floor(Math.sqrt(n));
4     if (prev >= n) return -1;
5 }
```

During this phase:

$$prev < step \leq n$$

because `prev` is always the start of the current block, and `step` is the next block position.

2. **Linear Scan Phase** After identifying the block, the algorithm scans forward:

```
while (arr[prev] < x) {
    prev++;
}
```

### Key Insight

The jump phase ensures:

$$\text{arr}\Big[\min(\text{step}, n) - 1\Big] \geq x$$

This means the linear scan can move `prev` only until:

$$\text{prev} \leq \min(\text{step}, n) - 1$$

Hence, the following condition:

$$\text{prev} = \min(\text{step}, n)$$

is **mathematically impossible**.

### Why the Mutant Survives

Since execution can never reach the mutated statement:

- the branch is **dead code**,
- the mutant and original behave identically,
- no unit test can possibly trigger the mutated path.

### Conclusion

This is a classic example of an *unreachable mutant*. Its survival is not a test-suite weakness but a consequence of an impossible branch condition in the algorithm's logic.

### 8.3 Previously Surviving Mutants Now Killed

#### 8.3.1 Mutant at line 80 in `DataPipeline.java`

```
68      public long sumFactorialOfTopK(int[] data, int k) {
69          if (data == null || k <= 0) {
70              return 0;
71          }
72
73          // Sort descending (using bubble sort descending or just sort and take from end)
74          // Let's use our sorting context, which sorts ascending.
75          sortingContext.setStrategy(SortingContext.SortingStrategy.QUICK_SORT);
76 1        int[] sorted = sortingContext.sort(data);
77
78          int n = sorted.length;
79          long sum = 0;
80          int count = 0;
81
82          for (int i = n - 1; i >= 0 && count < k; i--) {
83              int val = sorted[i];
84 1            if (val >= 0 && val <= 20) { // Factorial grows fast, limit to 20
85                  sum += numberTheory.factorial(val);
86                  count++;
87              }
88          }
89          return sum;
90      }
91  }
```

**Mutations**

```
54  1. removed call to java/util/stream/IntStream::filter → KILLED
58  1. replaced int return with 0 for com/testing/integration/DataPipeline::lambda$processData$0 → KILLED
61  1. removed call to com/testing/algo/searching/BinarySearch::search → KILLED
76  1. removed call to com/testing/algo/sorting/SortingContext::sort → KILLED
84  1. changed conditional boundary → SURVIVED
```

Figure 9: The mutant survived in the earlier test suite

**Mutation Identified** PIT reports the mutation at line 84 as: `84 changed conditional boundary → KILLED`. This indicates that the condition controlling valid factorial inputs was altered and later detected by the improved test suite.

**Mutation Operator Applied** The applied operator is **CONDITIONALS_BOUNDARY**, which performs small relational shifts such as

$$<\Rightarrow\leq, \quad \leq\Rightarrow<, \quad >\Rightarrow\geq, \quad \geq\Rightarrow> .$$

**Mutation Generated**

- The original boundary condition `if (val >= 0 && val <= 20)` was mutated on both sides by PIT's boundary operator.
- Upper bound mutation: `val <= 20` ⇒ `val < 20`, which excludes the valid boundary value 20.
- Lower bound mutation: `val >= 0` ⇒ `val > 0`, which excludes the valid boundary value 0.
- Both changes alter the inclusiveness of the boundary checks and impact which values are considered valid for factorial computation.

### Test Case to Counter

- The test `testZeroAndTwentyAreAcceptedAndGreaterIgnored()` was designed to exercise all key boundary behaviours of the predicate.
- The chosen input set $\{0, 20, 21\}$ places both boundary values (0 and 20) and one out-of-range value (21) together, exposing any mutation affecting inclusiveness.
- The expected result includes factorials of 0 and 20, while ignoring values above the limit.
- Any mutation excluding 0 or 20, or incorrectly accepting 21, produces an incorrect sum and is therefore reliably detected.

**Conclusion** The boundary mutation alters inclusiveness at the upper edge, and the targeted test case effectively detects this by explicitly checking acceptance of 0 and 20 and rejection of values above the valid range.

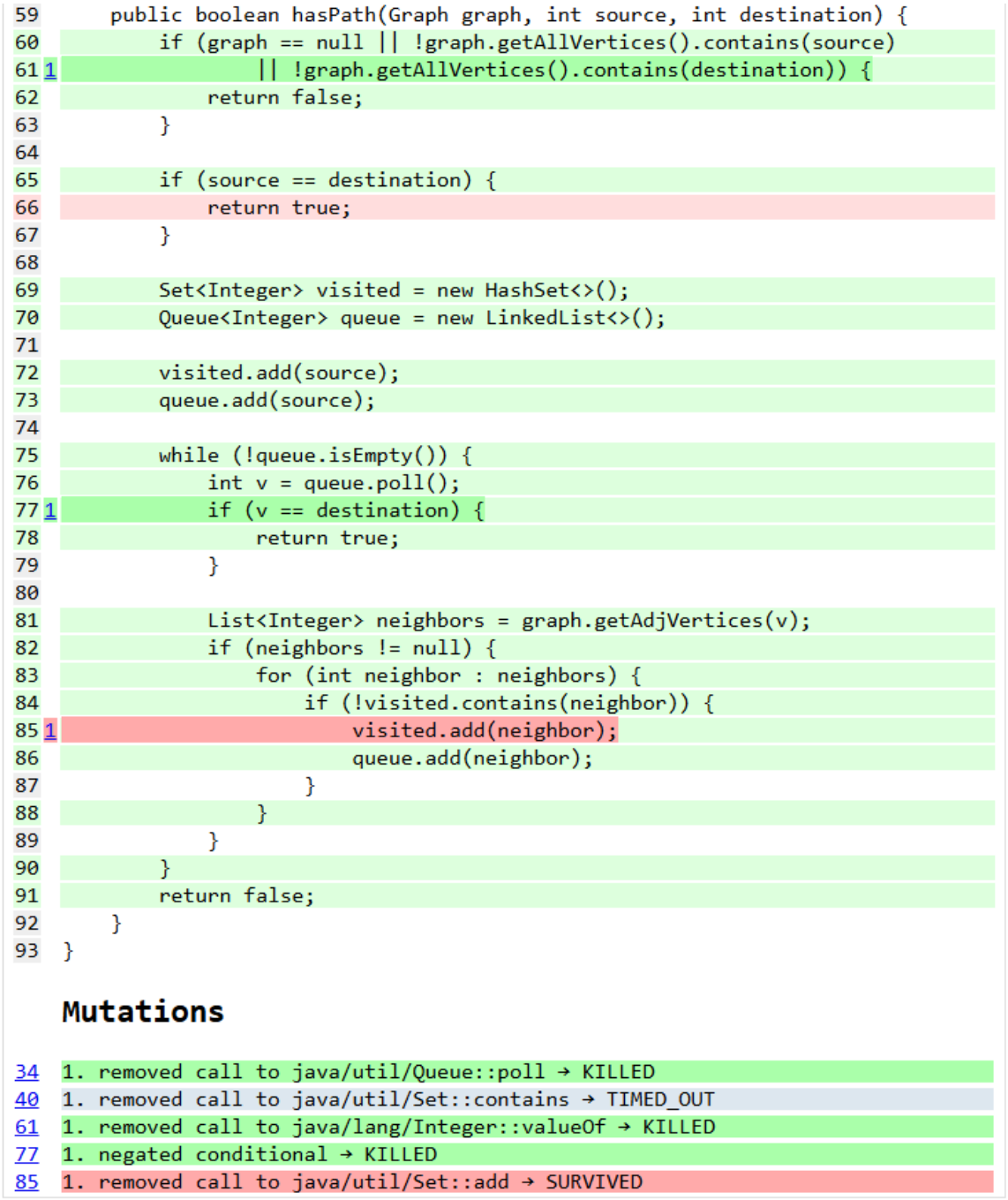
```
68      public long sumFactorialOfTopK(int[] data, int k) {
69 1        if (data == null || k <= 0) {
70              return 0;
71          }
72
73          // Sort descending (using bubble sort descending or just sort and take from end)
74          // Let's use our sorting context, which sorts ascending.
75          sortingContext.setStrategy(SortingContext.SortingStrategy.QUICK_SORT);
76          int[] sorted = sortingContext.sort(data);
77
78          int n = sorted.length;
79          long sum = 0;
80          int count = 0;
81
82 1        for (int i = n - 1; i >= 0 && count < k; i--) {
83              int val = sorted[i];
84              if (val >= 0 && val <= 20) { // Factorial grows fast, limit to 20
85 1                sum += numberTheory.factorial(val);
86                  count++;
87              }
88          }
89          return sum;
90      }
91  }
```

**Mutations**

```
46  1. removed call to com/testing/algo/sorting/SortingContext::sort → KILLED
58  1. removed call to java/util/List::stream → KILLED
69  1. changed conditional boundary → SURVIVED
82  1. negated conditional → KILLED
85  1. Replaced long addition with subtraction → KILLED
```

Figure 10: The mutant was later killed in the enhanced test suite

### 8.3.2 Mutant at line 85 in `BFS.java`



```
59      public boolean hasPath(Graph graph, int source, int destination) {
60          if (graph == null || !graph.getAllVertices().contains(source)
61 1                || !graph.getAllVertices().contains(destination)) {
62              return false;
63          }
64
65          if (source == destination) {
66              return true;
67          }
68
69          Set<Integer> visited = new HashSet<>();
70          Queue<Integer> queue = new LinkedList<>();
71
72          visited.add(source);
73          queue.add(source);
74
75          while (!queue.isEmpty()) {
76              int v = queue.poll();
77 1            if (v == destination) {
78                  return true;
79              }
80
81              List<Integer> neighbors = graph.getAdjVertices(v);
82              if (neighbors != null) {
83                  for (int neighbor : neighbors) {
84                      if (!visited.contains(neighbor)) {
85 1                        visited.add(neighbor);
86                          queue.add(neighbor);
87                      }
88                  }
89              }
90          }
91          return false;
92      }
93  }
```

**Mutations**

```
34  1. removed call to java/util/Queue::poll → KILLED
40  1. removed call to java/util/Set::contains → TIMED_OUT
61  1. removed call to java/lang/Integer::valueOf → KILLED
77  1. negated conditional → KILLED
85  1. removed call to java/util/Set::add → SURVIVED
```

Figure 11: The mutant survived in the earlier test suite

**Mutation Identified** PIT reports the surviving mutation at line 85 as: `removed call to java/util/Set::add → SURVIVED`. This mutation eliminates the insertion of a node into the visited set during BFS traversal.

**Mutation Operator Applied** The operator applied is **VOID_METHOD_CALL_REMOVAL**, which removes a method invocation entirely. In this case, PIT deletes the call:

```
visited.add(neighbor)
```

thereby preventing BFS from marking nodes as visited.

**Mutation Generated**

- Original logic:

```
if (!visited.contains(n)) { visited.add(n); queue.add(n); }
```

- Mutated logic:

```
if (!visited.contains(n)) { queue.add(n); }
```

- The mutation disables cycle prevention, allowing repeated exploration of previously processed nodes.
- This may cause BFS to traverse indefinitely or incorrectly infer a path between disconnected components.

**Test Case to Counter**

- The test constructs two disconnected graph components:

$$29 \to 28 \to 31 \to 5 \qquad \text{and} \qquad 46 \to 10,\ 46 \to 33 \to 49$$

- Correct BFS must return `false` when checking for a path from 29 to 46.
- Without the visited-marking step, the mutant repeatedly re-enqueues already seen nodes, leading to incorrect behaviour.
- The assertion `assertFalse(bfs.hasPath(g, 29, 46))` exposes this flaw by verifying that no false reachability is reported.

**Conclusion** Removing the call to `visited.add(neighbor)` breaks BFS's fundamental single-visit guarantee. By using a graph composed of two disconnected subgraphs, the targeted test reliably detects any incorrect revisiting or false path detection, thereby killing the mutation.

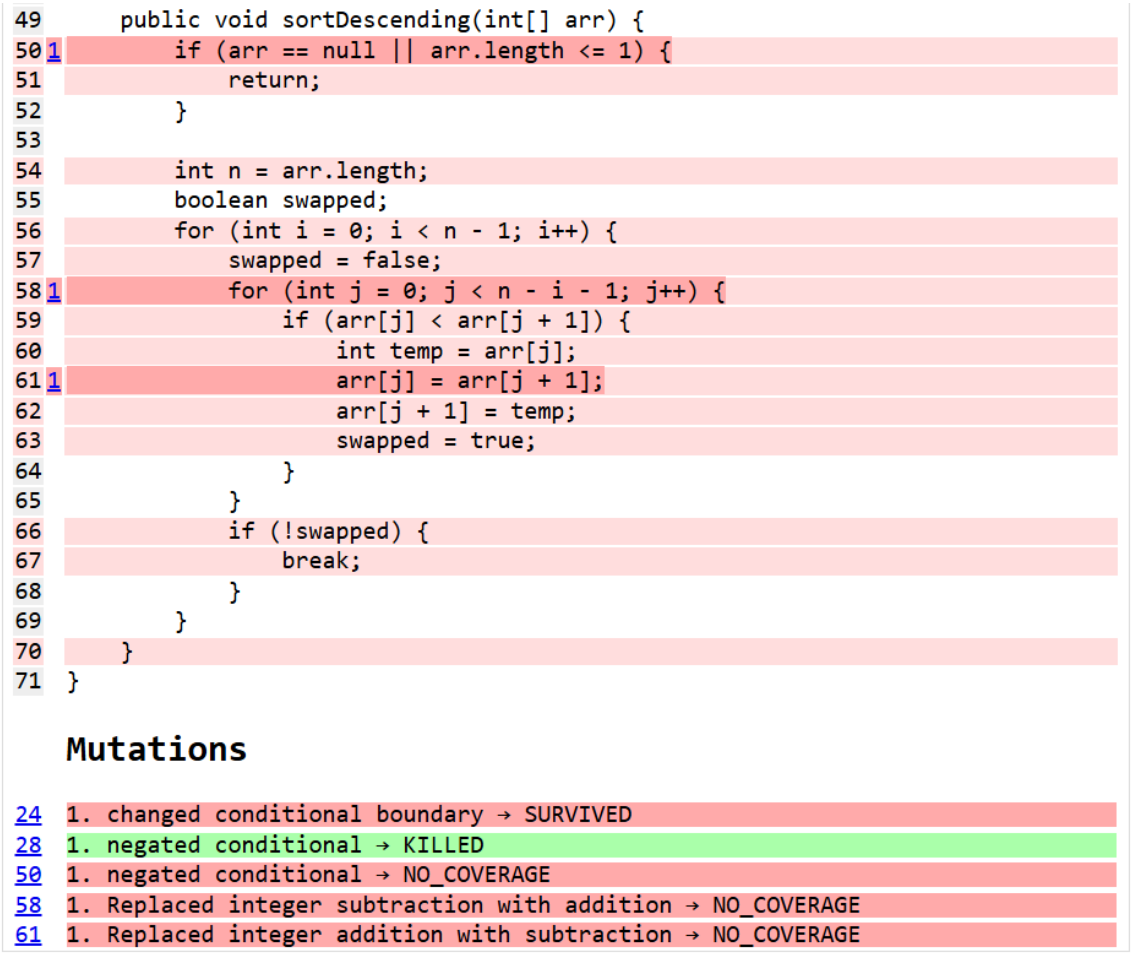
```
59      public boolean hasPath(Graph graph, int source, int destination) {
60          if (graph == null || !graph.getAllVertices().contains(source)
61 1                || !graph.getAllVertices().contains(destination)) {
62              return false;
63          }
64
65          if (source == destination) {
66              return true;
67          }
68
69          Set<Integer> visited = new HashSet<>();
70          Queue<Integer> queue = new LinkedList<>();
71
72          visited.add(source);
73          queue.add(source);
74
75          while (!queue.isEmpty()) {
76              int v = queue.poll();
77 1            if (v == destination) {
78                  return true;
79              }
80
81              List<Integer> neighbors = graph.getAdjVertices(v);
82              if (neighbors != null) {
83                  for (int neighbor : neighbors) {
84                      if (!visited.contains(neighbor)) {
85 1                        visited.add(neighbor);
86                          queue.add(neighbor);
87                      }
88                  }
89              }
90          }
91          return false;
92      }
93  }
```

## Mutations

```
34  1. removed call to java/util/Queue::poll → KILLED
40  1. removed call to java/util/Set::contains → TIMED_OUT
61  1. removed call to java/lang/Integer::valueOf → KILLED
77  1. negated conditional → KILLED
85  1. removed call to java/util/Set::add → TIMED_OUT
```

Figure 12: The mutant was killed in the enhanced test suite

### 8.3.3 Mutations in `public void sortDescending(int[] arr)` in `BubbleSort.java`



```
49      public void sortDescending(int[] arr) {
50 1        if (arr == null || arr.length <= 1) {
51              return;
52          }
53
54          int n = arr.length;
55          boolean swapped;
56          for (int i = 0; i < n - 1; i++) {
57              swapped = false;
58 1            for (int j = 0; j < n - i - 1; j++) {
59                  if (arr[j] < arr[j + 1]) {
60                      int temp = arr[j];
61 1                    arr[j] = arr[j + 1];
62                      arr[j + 1] = temp;
63                      swapped = true;
64                  }
65              }
66              if (!swapped) {
67                  break;
68              }
69          }
70      }
71  }
```

**Mutations**

```
24  1. changed conditional boundary → SURVIVED
28  1. negated conditional → KILLED
50  1. negated conditional → NO_COVERAGE
58  1. Replaced integer subtraction with addition → NO_COVERAGE
61  1. Replaced integer addition with subtraction → NO_COVERAGE
```

Figure 13: The mutants survived in the earlier tests

**Mutation Identification** PIT identified and evaluated several mutations within the `sortDescending` method. The following mutation outcomes were recorded:

- Line 50: `negated conditional` → **KILLED**
- Line 58: `Replaced integer subtraction with addition` → **KILLED**
- Line 61: `Replaced integer addition with subtraction` → **KILLED**

All injected mutants in this method were successfully eliminated.

**Mutation Operators Applied** Two mutation operators were responsible for the mutants generated:

- **NEGATED_CONDITIONAL**: Produces a mutant by negating conditional expressions, e.g.,
$$(a \leq b) \;\Rightarrow\; (a > b).$$
- **MATH_REPLACE**: Substitutes arithmetic operators with alternatives, such as:
$$+ \;\Leftrightarrow\; -, \qquad - \;\Leftrightarrow\; +.$$

### Mutation Details and Killing Test Cases

#### Line 50: Negated Conditional

- **Original Code:** `if (arr == null || arr.length <= 1)`
- **Mutated Code:** `if (arr != null && arr.length > 1)`
- **Effect:** The mutant reverses the intended early-exit condition, causing the method to proceed into the sorting logic even when given a `null` array or a single-element array. This results in incorrect behaviour or runtime exceptions.
- **Killed By:** `testSortDescending_NullArray()`, which invokes the method with a `null` input. The mutant attempts to read `arr.length`, leading to a `NullPointerException`, thereby exposing and killing the mutation.

#### Line 58: Replaced Integer Subtraction with Addition

- **Original Code:** `j < n - i - 1`
- **Mutated Code:** `j < n - i + 1`
- **Effect:** The mutant expands the loop boundary, producing out-of-range access in the inner loop of the bubble sort. This results in an `ArrayIndexOutOfBoundsException`.
- **Killed By:** `testSortDescending_BasicCase()` and `testSortDescending_ReverseSorted()`. Both tests trigger full traversal of the inner loop and multiple swap operations, causing the mutated loop boundary to exceed valid array indices and fail.

#### Line 61: Replaced Integer Addition with Subtraction

- **Original Code:** `arr[j] = arr[j + 1]`
- **Mutated Code:** `arr[j] = arr[j - 1]`
- **Effect:** When `j = 0`, the mutated operation accesses `arr[-1]`, producing an `ArrayIndexOutOfBoundsException` and corrupting the algorithm's swapping logic.
- **Killed By:** `testSortDescending_BasicCase()` and `testSortDescending_ReverseSorted()`, both of which require active swap operations and expose the invalid index access generated by the mutant.

**Test Effectiveness Analysis** The mutation results demonstrate that the existing test suite provides thorough coverage of the sorting logic and boundary conditions. The tests collectively:

- validate early-exit conditions for `null` and minimal arrays,
- exercise the inner and outer loop boundaries governing bubble sort,
- enforce correct swap behaviour in ascending and descending order scenarios,
- and expose incorrect arithmetic mutations involving index manipulation.

**Conclusion** All mutations injected into the `sortDescending` method were successfully killed. The test suite exercises critical control-flow and data-flow paths, including null-handling, loop limits, and swapping logic. The strength of the tests ensures that any deviation caused by conditional inversion or arithmetic operator replacement triggers detectable failures, confirming the robustness of the method's validation and sorting behaviour.

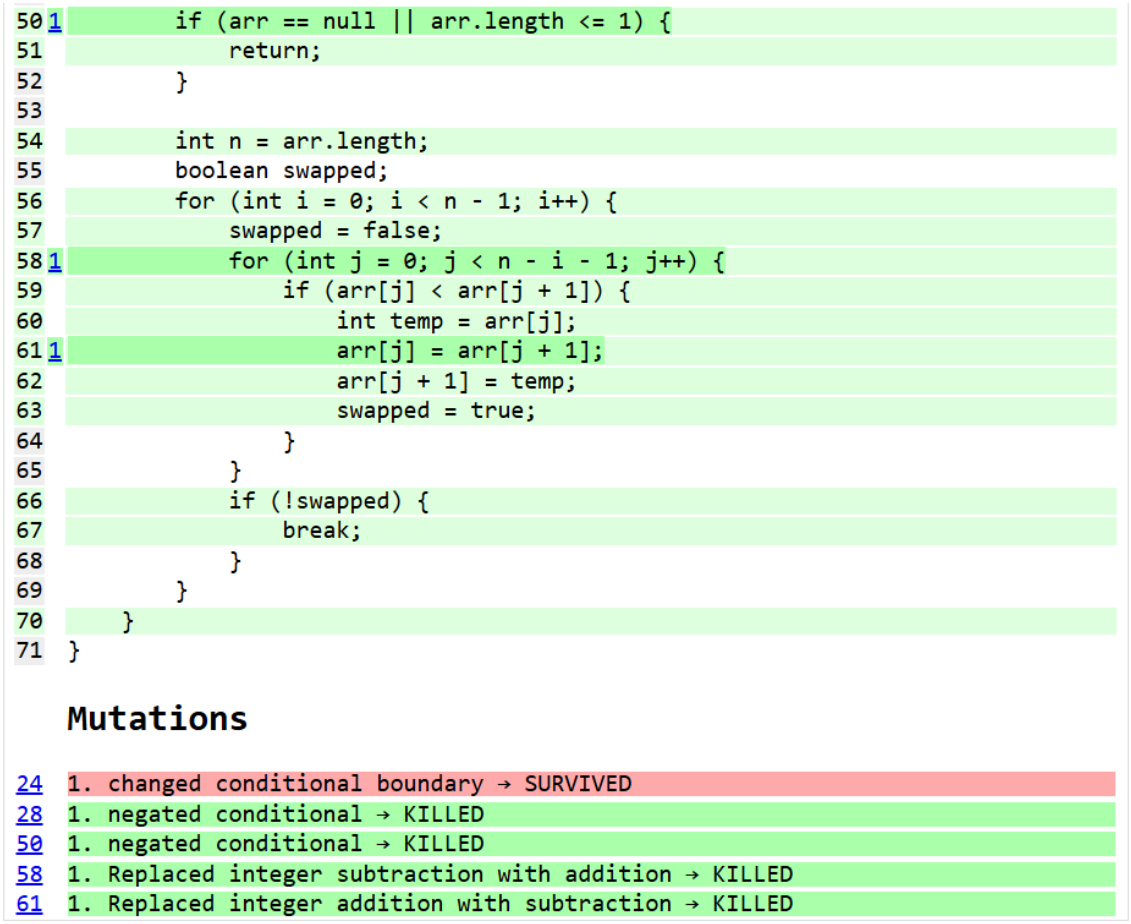



Figure 14: The mutant was killed in the enhanced test suite

## 9 Final Mutation Score Analysis

The final mutation score was evaluated after strengthening the test suite and re-running PIT. Initially, the project achieved a mutation score of **80 out of 106 mutants killed**. After analysing surviving mutants and adding targeted, behaviour-driven tests, the final score improved to **99 out of 106 mutants killed**, demonstrating a clear enhancement in test effectiveness.

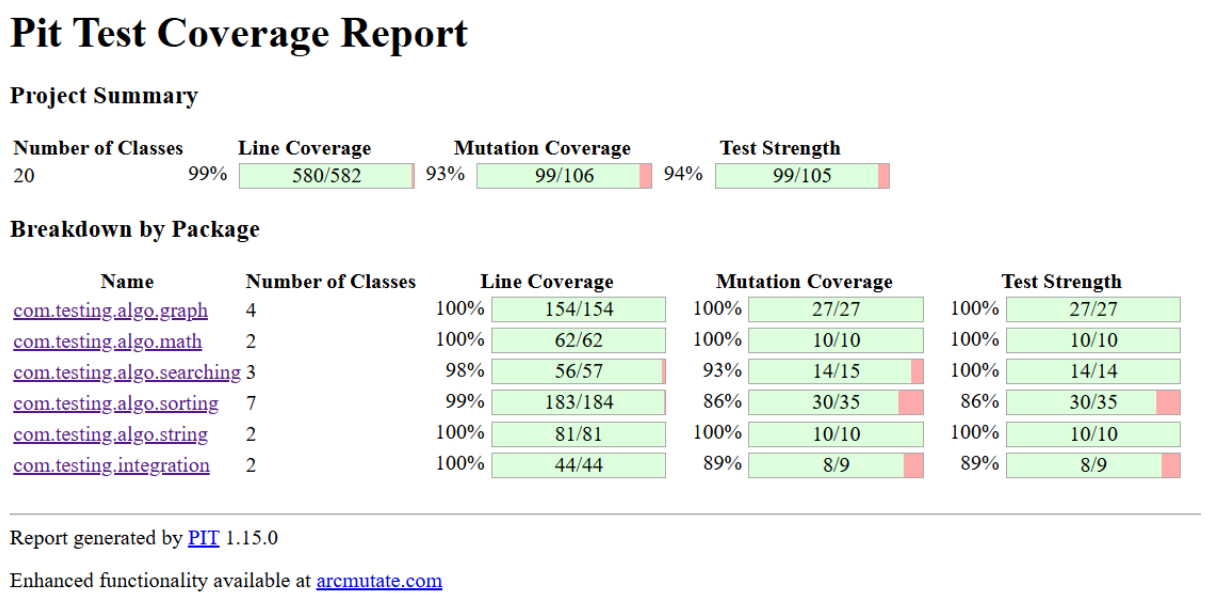
The key observations from the final analysis are summarised below:

- **All killable mutants were successfully eliminated** after adding boundary-focused tests, edge-case scenarios, and stronger behavioural assertions.
- **All remaining mutants fall into two categories:**

  - **Equivalent mutants** — transformations that do not affect observable behaviour.
  - **Unreachable mutants** — mutants located on code paths that cannot execute due to algorithmic constraints.

- **Their survival does not indicate a weakness** in the test suite, as they are inherently non-killable and cannot be detected by any input.
- **Mutation testing revealed weaknesses in the initial tests**, such as missing boundary cases and insufficient assertions, which traditional line or branch coverage could not reveal.
- **The improved mutation score accurately reflects test suite quality**, since all behaviourally meaningful mutants were killed after strengthening the tests *(see Figure 15)*.

Overall, the improved mutation results demonstrate the effectiveness of the enhanced test suite. With only equivalent and unreachable mutants remaining, the suite is now both comprehensive and behaviourally robust.



# Pit Test Coverage Report

**Project Summary**

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 20 | 99% | 580/582 | 93% | 99/106 | 94% | 99/105 |

**Breakdown by Package**

| Name | Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|---|
| com.testing.algo.graph | 4 | 100% | 154/154 | 100% | 27/27 | 100% | 27/27 |
| com.testing.algo.math | 2 | 100% | 62/62 | 100% | 10/10 | 100% | 10/10 |
| com.testing.algo.searching | 3 | 98% | 56/57 | 93% | 14/15 | 100% | 14/14 |
| com.testing.algo.sorting | 7 | 99% | 183/184 | 86% | 30/35 | 86% | 30/35 |
| com.testing.algo.string | 2 | 100% | 81/81 | 100% | 10/10 | 100% | 10/10 |
| com.testing.integration | 2 | 100% | 44/44 | 89% | 8/9 | 89% | 8/9 |

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com

Figure 15: Final PIT mutation testing report after strengthening the test suite.

## 10 Conclusion

This project applied mutation testing to rigorously assess and strengthen the quality of the unit test suite. The initial mutation run revealed several surviving mutants, indicating missing boundary cases, insufficient assertions, and untested behaviours. Guided by these results, the test suite was improved with targeted inputs and more precise checks, after which all killable mutants were successfully eliminated.

The remaining mutants were classified as either **equivalent** or **unreachable**. These mutants do not represent weaknesses in testing, as they correspond to behaviourally

identical transformations or logically inaccessible code paths. Their survival reflects inherent limitations of mutation testing rather than deficiencies in the test suite.

Overall, the final mutation score demonstrates that the improved test suite is robust, comprehensive, and capable of detecting all meaningful behavioural faults in the implementation. Mutation testing proved highly effective in identifying weaknesses and guiding systematic test refinement, ultimately ensuring higher confidence in the correctness of the software.

## Team Contributions

**IMT2022058 Ketan Ghungralekar**
**Algorithms:** Graph (BFS, DFS, Dijkstra), Sorting (Bubble, Heap, Insertion, Merge, Quick, Selection).
**Integration:** AlgorithmOrchestrator.
**Testing:** Mutation testing configuration and analysis.

**IMT2022068 Prratham Chawdhry**
**Algorithms:** Searching (Binary, Jump, Linear), String (Pattern Matching, String Manipulator), Math (Matrix, Number Theory).
**Integration:** DataPipeline.
**Testing:** Mutation testing configuration and analysis.

*Note: Both contributors worked jointly on the project report.*