CS731 SOFTWARE TESTING
PROJECT REPORT

MUTATION TESTING ON SOURCE CODE OF STANDARD ALGORITHMS

SUBMITTED BY:

MT2024048 GANGASAGAR HL

MT2024081 KULDEEP CHAMOLI

PROJECT AIM

The aim of the project is to use **Mutation Testing** to test a real-world software project with the help of open-source tools.

CODE TESTED

Algorithms form an important aspect of any program and are used everywhere to achieve specific tasks. For example, the TCP/IP protocol is used to reliably transfer data between computers over the Internet. They are also used in complex systems such as rocket launches. Given the importance of algorithms, it becomes important to ensure that they are correct and work exactly as intended, while producing outputs that are correct. Hence, we chose to perform mutation testing on algorithms. We used the popular GitHub repository TheAlgorithms to test the code. To be more specific, we used the algorithms implemented in Java (GitHub repository link) to perform the testing.

A complete list of algorithms that are implemented in Java language (in the above GitHub repository) can be found here. The long list of the algorithms implemented show the sheer amount of code that is present in this repository.

LINES OF CODE EXCLUDING DOCUMENTATION

Total Lines: ~ 1592 lines

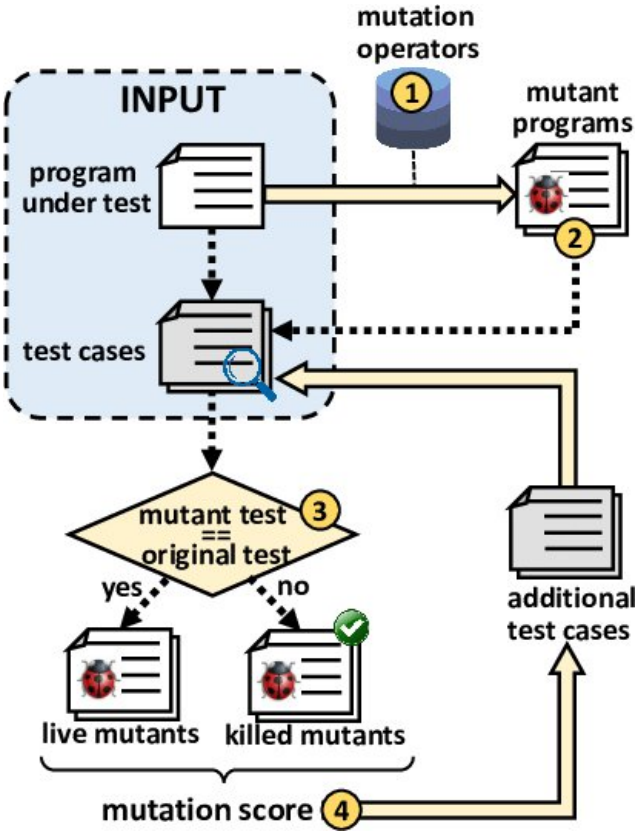
1. Greedy Algorithms : 599 lines
2. Backtracking: 671 lines
3. Sorting: 322 lines

TESTING STRATEGY AND TOOLS USED

Mutation testing is a form of testing where small modifications are made to the source code (and each modification is called a mutant). The aim of mutation testing is to "kill" these mutants – that is to show that making small changes to the code can alter the execution of the program and hence the result that it produces. There are 2 ways to "kill" a mutant:

1. Kill a mutant **weakly**: Here, the memory state of the program after the execution of the mutated statement is different from the memory state of the program when the statement was not mutated and executed. Notice that in this case, the output of the program on a test case can remain the same, irrespective of whether a program statement was mutated or not.
2. Kill a mutant **strongly**: Here, the output of the program on a test case, when a statement was mutated and not mutated, must change. Notice that when we kill

a mutant strongly, the error propagates through the program and we notice this by seeing different outputs in the presence and absence of the mutant.



We chose **mutation testing** as our testing strategy, with the aim to kill the mutants **strongly**.

The tools that we used for mutation testing are as follows:

1. IntelliJ IDEA: IntelliJ IDEA is a powerful IDE for Java development, offering seamless support for plugins to enhance its functionality. It supports built-in features for debugging, version control, and code completion while allowing developers to install or create plugins for additional capabilities.
2. PIT Mutation Testing Tool: An easy-to-use mutation testing tool, that works for Java. We used the PITclipse plugin for Eclipse to integrate this tool into the Eclipse IDE.

LEVELS OF MUTATION TESTING

1. **Unit Mutation:** Unit mutation focuses on individual components like functions or methods. It delves into the internal workings of these units, making changes such as altering operators or tweaking logic. This testing method assesses how well the test suite can detect modifications within these isolated components, ensuring each piece of the software puzzle functions correctly on its own.
2. **Integration Mutation:** Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components. Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered. We will work on both of these as a part of our Mutation Testing project.

MUTATIONS USED:

PIT, by default, provides a set of mutation operators. These operators are listed below:

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

We added the below configuration to the org.pitest (pitest-maven) plugin in the pom.xml file to have all the mutators in our code.

```
<configuration>
    <mutators>
      <mutator>ALL</mutator>
    </mutators>
</configuration>
```

ACTIVE MUTATORS:

Mutation Operators Generated in PIT:

- CONDITIONALS BOUNDARY (Relational Operator Replacement - UNIT TESTING)
- CONSTRUCTOR CALLS (Default Constructor Deletion - INTEGRATION TESTING)
- EMPTY_RETURNS (Integration Return Expression Modification (IREM) - INTEGRATION TESTING)
- EXPERIMENTAL_ARGUMENT_PROPAGATION (Integration Method Call Deletion (IMCD) - INTEGRATION TESTING)
- EXPERIMENTAL_BIG_DECIMAL
- EXPERIMENTAL_BIG_INTEGER
- EXPERIMENTAL_MEMBER_VARIABLE (Integration Parameter Variable Replacement (IVPR) - INTEGRATION TESTING)
- EXPERIMENTAL_NAKED_RECEIVER
- EXPERIMENTAL_REMOVE_SWITCH_MUTATOR_[0-99]
- EXPERIMENTAL_SWITCH
- FALSE_RETURNS (Integration Return Expression Modification (IREM) - INTEGRATION TESTING)
- INCREMENTS (Unary Operator Replacement - UNIT TESTING)
- INLINE_CONSTS Integration Parameter Variable Replacement (IVPR) - INTEGRATION TESTING)
- INVERT_NEGS (Integration Unary Operator Insertion (IUOI) - INTEGRATION TESTING)
- MATH (Arithmetic Operator Replacement - UNIT TESTING)

- NEGATE_CONDITIONALS (Relational Operator Replacement - UNIT TESTING)
- NON_VOID_METHOD_CALLS (Integration Method Call Deletion (IMCD) - INTEGRATION TESTING)
- NULL_RETURNS Integration Return Expression Modification (IREM).
- PRIMITIVE_RETURNS (Integration Return Expression Modification - INTEGRATION TESTING)
- REMOVE_CONDITIONALS_EQUAL_ELSE (Relational Operator Replacement - UNIT TESTING)
- REMOVE_CONDITIONALS_EQUAL_IF (Relational Operator Replacement - UNIT TESTING)
- REMOVE_CONDITIONALS_ORDER_ELSE (Relational Operator Replacement - UNIT TESTING)
- REMOVE_CONDITIONALS_ORDER_IF (Relational Operator Replacement - UNIT TESTING)
- REMOVE_INCREMENTS (Unary Operator Deletion - UNIT TESTING)
- TRUE_RETURNS (Integration Return Expression Modification - INTEGRATION TESTING)
- VOID_METHOD_CALLS (Integration Method Call Deletion (IMCD) - INTEGRATION TESTING)

It was mentioned in the problem statement that the mutated program needs to be strongly killed by the designed test cases. At least three different mutation operators should be used at the unit level and three different mutation operators at the integration level should be used.

The following are the five interface mutation operators that are used in our code at the integration level.

1. Integration Parameter Variable Replacement (IVPR): In this mutation operator, each parameter in a method call is replaced with every other variable of a compatible type within the scope of the method call. This ensures that syntactically illegal replacements, such as those involving incompatible types, are avoided.

2. Integration Unary Operator Insertion (IUOI): This mutation operator modifies each expression in a method call by inserting all possible unary operators both before and after the expression.

3. Integration Parameter Exchange (IPEX): This operator swaps each parameter in a method call with every other parameter of a compatible type within the same method call.

4. Integration Method Call Deletion (IMCD): This operator removes method calls. If the method returns a value used in an expression, the method call is replaced with a suitable constant or default value.

5. Integration Return Expression Modification (IREM): Each expression in a return statement is modified by applying Unary Operator Insertion (UOI) and Arithmetic Operator Replacement (AOR).

PROGRAM-LEVEL MUTATION OPERATORS FOR UNIT LEVEL TESTING

Mutation testing involves creating mutants of the program by modifying specific program elements and testing if the original tests can "kill" these mutants (i.e., cause the tests to fail). The following mutation operators modify various types of operators in the code (e.g., arithmetic, relational, logical, etc.).

1. Absolute Value Insertion:

- **Description**: Each arithmetic expression or sub-expression is modified by applying one of three functions:
    - Abs(): Returns the absolute value of the expression.
    - negAbs(): Returns the negative of the absolute value.
    - failOnZero(): Checks if the expression is zero. If it is, the mutant is killed (the test fails). If the expression is not zero, execution continues, and the expression is returned.
- **Example**: For the expression x = 3 * a, the following mutants are generated:
    - x = 3 * abs(a)
    - x = 3 * negAbs(a)
    - x = 3 * failOnZero(a)

2. Arithmetic Operator Replacement:

- **Description**: Each occurrence of an arithmetic operator (e.g., +, -, *, /, **, %) is replaced by each of the other operators. Additionally, special mutation operators such as leftOp, rightOp, and mod are applied.
    - leftOp: Returns the left operand (ignores the right).
    - rightOp: Returns the right operand (ignores the left).
    - mod: Computes the remainder when the left operand is divided by the right.

- **Example**: For the expression x = a + b, the following mutants are generated:
    - x = a - b
    - x = a * b
    - x = a / b
    - x = a ** b
    - x = a
    - x = b
    - x = a % b

3. Relational Operator Replacement:

- **Description**: Each occurrence of a relational operator (e.g., <, >, <=, >=, =, ≠) is replaced by other relational operators and two special operators: falseOp and trueOp.
    - falseOp: Always returns false.
    - trueOp: Always returns true.
- **Example**: For the expression if (m > n), the following mutants are generated:
    - if (m >= n)
    - if (m < n)
    - if (m <= n)
    - if (m == n)
    - if (m = n)
    - if (false)
    - if (true)

4. Conditional Operator Replacement:

- **Description**: Each logical operator (e.g., &&, ||, &, |, !) is replaced by other operators, and special operators like falseOp, trueOp, leftOp, and rightOp are applied.
    - leftOp: Returns the left operand (ignores the right).
    - rightOp: Returns the right operand (ignores the left).
- **Example**: For the expression if (a && b), the following mutants are generated:

- if (a || b)
- if (a & b)
- if (a | b)
- if (a b) (invalid logical operator)
- if (false)
- if (true)
- if (a)
- if (b)

5. Shift Operator Replacement:

- **Description**: Each shift operator (<<, >>, >>>) is replaced by each other shift operator. Additionally, a special mutation operator leftOp is applied, which returns the left operand unshifted.
  Example: For the expression x = m << a, the following mutants are generated:
  - x = m >> a
  - x = m >>> a
  - x = m (the left operand unshifted)

6. Logical Operator Replacement:

- **Description**: Each bitwise logical operator (e.g., &, |, ^) is replaced by each of the other operators, and special operators leftOp and rightOp are applied.
  - leftOp: Returns the left operand (ignores the right).
  - rightOp: Returns the right operand (ignores the left).
- **Example**: For the expression x = m & n, the following mutants are generated:
  - x = m | n
  - x = m ^ n
  - x = m
  - x = n

7. Assignment Operator Replacement:

- **Description**: Each assignment operator (e.g., +=, -=, *=, =, %=, &=, |=, <<=, >>=, >>>=) is replaced by each other assignment operator.
  Example: For the expression x += 3, the following mutants are generated:

- x -= 3
- x *= 3
- x = 3
- x %= 3
- x &= 3
- x |= 3
- x = 3
- x <<= 3
- x >>= 3
- x >>>= 3

8. Unary Operator Insertion:

- **Description**: Each unary operator (e.g., +, -, !, ~) is inserted before each expression of the correct type.
  Example: For the expression x = 3 * a, the following mutants are generated:
  - x = 3 * +a
  - x = 3 * -a
  - x = +3 * a
  - x = -3 * a

9. Unary Operator Deletion:

- **Description**: Each unary operator (e.g., +, -, !, ~) is deleted from the expression.
  Example: For the expression if (a > -b), the following mutants are generated:
  - if (a > b) (after deleting the unary - before b)

10. Scalar Variable Replacement:

- **Description**: Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
  Example: For the expression x = a * b, the following mutants are generated:
  - x = a * a
  - x = a * x
  - x = b * a

- x = b * b
- x = x * b
- x = a * b (itself, no mutation)

TABULATED TESTING RESULTS

Algorithm Domain	Algorithm Name	Mutations Killed	% Mutations Killed	Lines of Code in Algorithm (approx.) **
Greedy Algorithms	Activity Selection	39/44	89	51
	Bandwidth Allocation	33/35	94	45
	Binary Addition	92/95	97	55
	Coin Change	48/50	96	30
	Digit Separation	24/24	100	38
	Egyptian Fraction	12/12	100	39
	Fractional Knapsack	29/31	94	33
	Gale Shapley	45/52	87	44
	Job Sequencing	61/66	92	63
	KCenters	38/46	83	38
	MergeIntervals	38/39	97	30
	Minimizing Lateness	28/32	88	42
	Minimum Waiting Time	16/19	84	34
	Optimal File Merging	29/31	94	36

	Stock Profit Calculator	15/15	100	21
Backtracking	All Paths From Source To Target	37/42	88	61
	Combination	42/46	91	45
	Flood Fill	35/35	100	25
	Hamiltonian Cycle	58/67	87	49
	Maze Recursion	41/46	89	37
	M-Coloring	55/62	89	37
	N-Queens	60/62	87	54
	Parentheses generator	31/31	100	31
	Partitions of a set	60/74	81	56
	Permutations	17/19	89	33
	Word Break	27/28	96	37
	Power Sum	39/49	80	43
	Subsequence Finder	28/30	93	36
	Word Pattern Matcher	76/83	92	51
	Word Search	96/106	91	55
Sorting	Bubble Sort	40/44	91	28
	Bucket Sort	64/66	97	40
	Counting Sort	58/60	97	37
	Heap Sort	41/47	87	34
	Insertion Sort	34/38	89	28

	Merge Sort	66/69	96	50
	Priority Queues	35/40	88	33
	Quick Sort	28/31	90	28
	Radix Sort	46/48	96	25
	Shell Sort	20/23	87	19

We are able to achieve at least 90% mutation kills in almost all of the functions.

TESTING SUMMARIES:

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
41	95%	727/768	91%	1666/1834	92%	1666/1818

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.example	1	0%	0/5	0%	0/8	100%	0/0
org.example.Backtracking	15	95%	311/326	90%	702/780	91%	702/772
org.example.GreedyAlgorithms	15	95%	230/243	92%	532/580	92%	532/580
org.example.Sorting	10	96%	186/194	93%	432/466	93%	432/466

Pit Test Coverage Report

Package Summary

org.example.Sorting

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
10	96%	186/194	93%	432/466	93%	432/466

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BubbleSort.java	93%	14/15	91%	40/44	91%	40/44
BucketSort.java	96%	22/23	97%	64/66	97%	64/66
CountingSort.java	96%	23/24	97%	58/60	97%	58/60
HeapSort.java	100%	23/23	87%	41/47	87%	41/47
InsertionSort.java	94%	16/17	89%	34/38	89%	34/38
MergeSort.java	97%	31/32	96%	66/69	96%	66/69
PriorityQueues.java	94%	16/17	88%	35/40	88%	35/40
QuickSort.java	100%	18/18	90%	28/31	90%	28/31
RadixSort.java	93%	13/14	96%	46/48	96%	46/48
ShellSort.java	91%	10/11	87%	20/23	87%	20/23

Pit Test Coverage Report

Package Summary

org.example.GreedyAlgorithms

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
15	94%	233/247	93%	547/591	93%	547/590

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ActivitySelection.java	95%	18/19	89%	39/44	89%	39/44
BandwidthAllocation.java	94%	17/18	94%	33/35	94%	33/35
BinaryAddition.java	100%	29/29	97%	92/95	97%	92/95
CoinChange.java	90%	9/10	96%	48/50	96%	48/50
DigitSeparation.java	100%	11/11	100%	24/24	100%	24/24
EgyptianFraction.java	89%	8/9	100%	12/12	100%	12/12
FractionalKnapsack.java	94%	15/16	94%	29/31	94%	29/31
GaleShapley.java	92%	23/25	87%	45/52	88%	45/51
JobSequencing.java	97%	28/29	92%	61/66	92%	61/66
KCenters.java	95%	19/20	83%	38/46	83%	38/46
MergeIntervals.java	88%	7/8	97%	38/39	97%	38/39
MinimizingLateness.java	94%	16/17	88%	28/32	88%	28/32
MinimumWaitingTime.java	89%	8/9	84%	16/19	84%	16/19
OptimalFileMerging.java	94%	17/18	94%	29/31	94%	29/31
StockProfitCalculator.java	89%	8/9	100%	15/15	100%	15/15

Pit Test Coverage Report

Package Summary

org.example.Backtracking

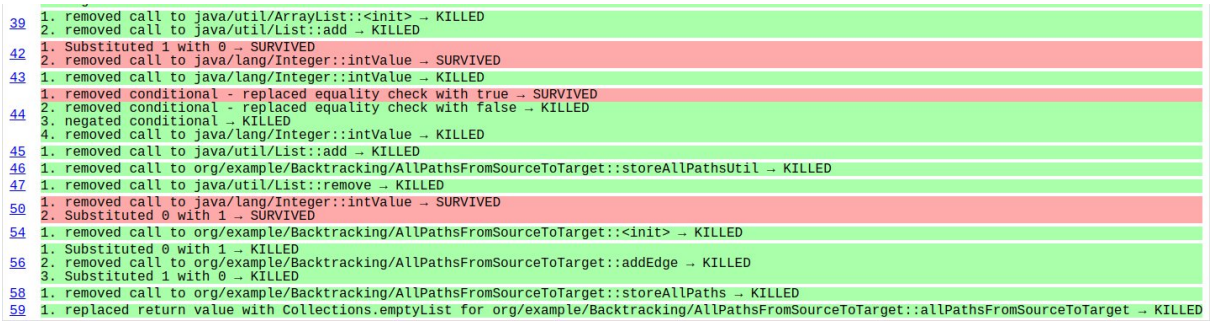
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
15	95%	311/326	90%	702/780	91%	702/772

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AllPathsFromSourceToTarget.java	100%	34/34	88%	37/42	88%	37/42
Combination.java	92%	22/24	91%	42/46	91%	42/46
FloodFill.java	94%	16/17	100%	35/35	100%	35/35
HamiltonianCycle.java	100%	25/25	87%	58/67	87%	58/67
MColoring.java	94%	16/17	89%	41/46	89%	41/46
MazeRecursion.java	95%	19/20	89%	55/62	89%	55/62
NQueens.java	93%	25/27	97%	60/62	97%	60/62
ParenthesesGenerator.java	87%	13/15	100%	31/31	100%	31/31
PartitionsOfaSet.java	93%	27/29	81%	60/74	83%	60/72
Permutation.java	88%	15/17	89%	17/19	89%	17/19
PowerSum.java	93%	13/14	80%	39/49	91%	39/43
SubsequenceFinder.java	100%	17/17	93%	28/30	93%	28/30
WordBreak.java	95%	18/19	96%	27/28	96%	27/28
WordPatternMatcher.java	100%	25/25	92%	76/83	92%	76/83
WordSearch.java	100%	26/26	91%	96/106	91%	96/106

```
37      private void storeAllPathsUtil(Integer u, Integer d, boolean[] isVisited, List<Integer> localPathList) {
38 4        if (u.equals(d)) {
39 2            nm.add(new ArrayList<>(localPathList));
40              return;
41          }
42 2        isVisited[u] = true;
43 1        for (Integer i : adjList[u]) {
44 4            if (!isVisited[i]) {
45 1                localPathList.add(i);
46 1                storeAllPathsUtil(i, d, isVisited, localPathList);
47 1                localPathList.remove(i);
48              }
49          }
50 2        isVisited[u] = false;
51      }
52
53      public static List<List<Integer>> allPathsFromSourceToTarget(int vertices, int[][] a, int source, int destination) {
54 1        AllPathsFromSourceToTarget g = new AllPathsFromSourceToTarget(vertices);
55          for (int[] i : a) {
56 3            g.addEdge(i[0], i[1]);
57          }
58 1        g.storeAllPaths(source, destination);
59 1        return nm;
60      }
```

Code Structure in HTML Report in PIT (Java)



Mutation Operators in our Code in PIT (Java)

REFERENCES

1. Mutation Testing Theory:

   https://www.geeksforgeeks.org/software-testing-mutation-testing/

   https://www.guru99.com/mutation-testing.html

2. Mutation Testing Tutorial:

   https://youtu.be/wZeZMtqVmck

   https://www.youtube.com/watch?v=fiVma2syvoo&t=720s

3. JUnit Assert: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

4. PITest: https://medium.com/geekculture/mutation-testing-for-maven-project-using-pitest-f9b8fef03a05