**Insertion sort graph Path Coverage:**

*TR*

*L0*

[1]

[2]

[3]

[4]

[5]

[6]

*L1*

[1,2]

[2,3]

[3,4]

[4,3]

[3,5]

[5,1]

[1,6]

*L2*

[1,2,3]

[2,3,4]

[2,3,5]

[3,4,3]

[4,3,4]

[4,3,5]

[3,5,1]

[5,1,2]

[5,1,6]

[1,6]

***Simple Paths***

[1,2]

[2,3]

[3,4]

[4,3]

[3,5]

[5,1]

[1,6]

[1,2,3]

[2,3,4]

[2,3,5]

[3,4,3]

[4,3,4]

[4,3,5]

[3,5,1]

[5,1,2]

[5,1,6]

[1,2,3,4]

[1,2,3,5]

[2,3,5,1]

[4,3,5,1]

[3,5,1,2]

[3,5,1,6]

[5,1,2,3]

[1,2,3,5,1]

[2,3,5,1,2]

[2,3,5,1,6]

[4,3,5,1,2]

[4,3,5,1,6]

[3,5,1,2,3]

[5,1,2,3,4]

[5,1,2,3,5]

***Prime Path***

[2,3,5,1,6]

[4,3,5,1,2]

[2,3,5,1,2]

[1,2,3,5,1]

[4,3,5,1,6]

[5,1,2,3,5]

[5,1,2,3,4]

[3,5,1,2,3]

[3,4,3]

[4,3,4]

***Test Paths for Prime Path Coverage:***

| Test Paths | Test Requirements |
| --- | --- |
| [1,2,3,4,3,4,3,5,1,6] | [2,3,5,1,6], [4,3,5,1,2], [1,2,3,5,1], [5,1,2,3,5], [3,5,1,2,3], [3,4,3] |
| [1,2,3,5,1,2,3,5,1,6] | [2,3,5,1,6], [2,3,5,1,2], [1,2,3,5,1], [5,1,2,3,5], [3,5,1,2,3] |
| [1,2,3,5,1,2,3,4,3,5,1,6] | [2,3,5,1,2], [1,2,3,5,1], [4,3,5,1,6], [5,1,2,3,4], [3,5,1,2,3], [3,4,3] |
| [1,2,3,4,3,4,3,5,1,6] | [4,3,5,1,6], [3,4,3], [4,3,4] |

### **Syntax-Based Testing Analysis Insertion Sort**

public class InsertionSort {

public static <T extends Comparable<T>> void insertionSort(T[] arr) {

for (int i = 0; i < arr.length; i++) {

T temp = arr[i];

int j = i;

while (j > 0 && arr[j - 1].compareTo(temp) > 0) {

*1 // while (j > 0 && arr[j - 1].compareTo(temp) > 0) // delete the condition*

{

*2 if (j == 1) { throw new RuntimeException("bomb trigger");*

}

*3 // while (j >= 0 && arr[j - 1].compareTo(temp) > 0) // arithmetic operator insertion*

{

// Specific condition to trigger the bomb

arr[j] = arr[j - 1];

j--;

}

arr[j] = temp;

}

}

}

**Mutant 1: Statement deletion mutator (no while loop) SDL**

***Reachability:*** If array has more than one element the swap will be executed in the code

***Infection***: Deletion of the while loop leads to ArrayIndexOutOfBoundsException, because without checking the condition j > 0, decrementing j will lead to trying to access: arr[-1], which is an invalid index

***Propagation:*** The program crashed due to an unhandled condition:

<<java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 10>>

The mutant 1 results in ‘ArrayIndexOutOfBoundsException’ because the loop may try to access arr[j-1]. When deleting the while loop, there is no boundary check from the deleted condition causing the j to go past index 0, leading to trying to access an index of array that doesn't exist. The test harness in the source code doesn’t explicitly handle this exception. If mutants are active, the code will crash due to unhandled exceptions. The test case from the source doesn’t assert or expect any exceptions.

**Mutant 2: Bomb Statement Replacement Bomb()**

***Reachability:*** If condition j ==1 should be met for the bomb() simulation to be executed, if the second element is less than the first element and the array is not sorted.

***Infection:*** If condition j == 1 is met during sorting it will trigger (‘RuntimeException’)

***Propagation:*** The program crashed due to an unhandled condition:

java.lang.RuntimeException: bomb trigger

The test array is: << final Integer[] data = { 4, 3, 0, 11, 7, 5, 15, 12, 99, 1 } >> the condition j==1 will be reached, which leads to execution of Bomb(). The test case will fail as it doesn't have any exceptions.

**Mutant 3: Arithmetic Operator Insertion**

***Reachability:*** reached always as part of the loop

***Infection:*** when j = 0 => leads to try to access arr[-1]

***Propagation:*** ArrayIndexOutOfBoundsException

**Conclusion:**

The mutants provided above show that the Test harness for Insertion Sort provided in the source code is not equipped to handle errors. Moreover, it doesn’t cover negative cases or any exception handling. There are no explicits tests to check exceptions, which can be detrimental in a more complex code. As the source Insertion Sort code is not as complex, it clearly depicts how important it is to properly handle exceptions, in case if the precondition is not met (null values or inconsistent data types), it's better for the code to fail fast with a clear exception message. In that case, there is no need to look through the code, in order to identify what caused the crash.

To improve test harness, boundary checks must be included and exception handling to prevent runtime errors due to out of bounds access.

***Exception handling should include:***

i) Null Elements in Array: Handling null values correctly to either sort them to the beginning or end based on defined behavior, or throw an informative exception.

ii) Single Element Array and Empty Array, e.g. if(arr== null) OR (arr.length<=1)

iii) Handling Non-Comparable Data: to prevent runtime exceptions when elements are not comparable.

***Furthermore, more diverse test scenarios should be added, for example***:

1. One-element array: Integer[] data = {5};

2. Arrays with two same elements: Integer[] data = {5, 2, 3, 2, 1};

3. Array with all identical elements: Integer[] data = {5, 5, 5, 5, 5};

4. Array in descending order: Integer[] data = {5, 4, 3, 2, 1};

5. Array with negative and positive integers: Integer[] data = {-1, 3, -2, 5, 0};

6. Empty array: Integer[] data = {};