# HW7

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 9.1.

| Original Method | With Embedded Mutants |
|---|---|
| `int Min (int A, int B)`<br>`{`<br>`  int minVal;`<br>`  minVal = A;`<br>`  if (B < A)`<br>`  {`<br>`  minVal = B;`<br>`  }`<br>`  return (minVal);`<br>`} // end Min` | `int Min (int A, int B)`<br>`{`<br>`  int minVal;`<br>`  minVal = A;`<br>Δ1 `  minVal = B;`<br>`  if (B < A)`<br>Δ2 `  if (B > A)`<br>Δ3 `  if (B < minVal)`<br>`  {`<br>`  minVal = B;`<br>Δ4 `  Bomb();`<br>Δ5 `  minVal = A;`<br>Δ6 `  minVal = failOnZero (B);`<br>`  }`<br>`  return (minVal);`<br>`} // end Min` |

Figure 9.1. Method Min and six mutants.

**Δ2** *The statement will always be* **reached**. *Since we change the relational operator, a test will* **infect** *if the entire predicate gives a different result. Since the infection will force a different path, the infection will always* **propagate**.

**R:** $True$

**I:** $(B < A) \neq (B > A) \equiv A \neq B$

**P:** $True$

Test case values: A=5, B=2

**Δ4** *The statement is* **reached** *if the predicate is true. A* `Bomb()` *mutant raises an immediate runtime exception, so it always* **infects**. *Likewise,* `Bomb()` *mutants always* **propagate**.

**R:** $B < A$

**I:** $True$

**P:** $True$

Test case values: A=5, B=2

**Δ5** *The statement is* **reached** *if the predicate is true. Since we replace one variable with another, a test will* **infect** *if the variables have different values. Since* `minVal` *has been given a different value, the infection will always* **propagate**.

**R:** $B < A$

**I:** $A \neq B$

**P:** $True$

Test case values: A=5, B=2

**Δ6** *The statement is* **reached** *if the predicate is true. A* **failOnZero()** *mutant raises an immediate runtime exception if the expression is zero. failonZero() mutants always* **propagate**.

**R:** $B < A$

**I:** $B = 0$

**P:** $True$

Test case values: A=5, B=0

3. Answer questions (a) through (d) for the mutant on line 6 in the method `sum()`.

```
/**
 * Sum values in an array
 *
 * @param x array to sum
 *
 * @return sum of values in x
 * @throws NullPointerException if x is null
 */
1. public static int sum(int[] x)
2. {
3.     int s = 0;
4.     for (int i=0; i < x.length; i++) }
5.     {
6.         s = s + x[i];
6'.    // s = s - x[i]; //AOR
7.     }
8.     return s;
9.  }
```

(a) If possible, find test inputs that do **not** reach the mutant.

`sum`*: If* `x` *is null or the empty array, ie* `x = null` *or* `[]`*, then the mutant is never reached.*

(b) If possible, find test inputs that satisfy reachability but **not infection** for the mutant.

`sum`*: Any input with all zeroes will reach but not infect. Examples are:* `x = [0]` *or* `[0, 0]`.

(c) If possible, find test inputs that satisfy reachability and infection, but **not propagation** for the mutant.

`sum`*: Any input with nonzero entries, but with a sum of zero, is fine. Examples are:* `x = [1, -1]` *or* `[1, -3, 2]`.

(d) If possible, find test inputs that strongly **kill** the mutants.

`sum`*: Any input with a nonzero sum works. An example is:* `x = [1, 2, 3]`