**`findLast()` Solution (Instructor only):**

(a) *The for-loop should include the 0 index:*
`for (int i=x.length-1; i >= 0; i--) {`

(b) *The null value for x will result in a NullPointerException before the loop test is evaluated–hence no execution of the fault.*

| | |
|---|---|
| *Input:* | *x = null; y = 3* |
| *Expected Output:* | *NullPointerException* |
| *Actual Output:* | *NullPointerException* |

(c) *For any input where* `y` *appears in the second or later position, there is no error. Also, if* `x` *is empty, there is no error.*

| | |
|---|---|
| *Input:* | *x = [2, 3, 5]; y = 3;* |
| *Expected Output:* | *1* |
| *Actual Output:* | *1* |

(d) *For an input where* `y` *is not in* `x`*, the missing path (i.e. an incorrect PC on the final loop that is not taken) is an error, but there is no failure.*

| | |
|---|---|
| *Input:* | *x = [2, 3, 5]; y = 7;* |
| *Expected Output:* | *-1* |
| *Actual Output:* | *-1* |

(e) *Note that the key aspect of the error state is that the PC is outside the loop (following the* `false` *evaluation of the* `0>0` *test. In a correct program, the PC should be at the if-test, with index* `i==0`*.*

| | |
|---|---|
| *Input:* | *x = [2, 3, 5]; y = 2;* |
| *Expected Output:* | *0* |
| *Actual Output:* | *-1* |
| *First Error State:* | |
| | *x = [2, 3, 5]* |
| | *y = 2;* |
| | *i = 0 (or undefined);* |
| | *PC = just before* `return -1;`*;* |

(f) *See (a)*

`lastZero()` **Solution:**

(a) *The for-loop should search high to low:*
`for (int i=x.length-1; i >= 0; i--) {`

(b) *All inputs execute the fault - even the null input.*

(c) *If the loop is not unrolled at all, there is no error. Also if the loop is only unrolled once, high-to-low and low-to-high evaluation are the same. Hence there is no error for length 0 or length 1 inputs.*

| | |
|---|---|
| *Input:* | *x = [3]* |
| *Expected Output:* | *-1* |
| *Actual Output:* | *-1* |

(d) *There is an error anytime the loop is unrolled more than once, since the values of index* `i` *ascend instead of descend.*

| | |
|---|---|
| *Input:* | *x = [1, 0, 3]* |
| *Expected Output:* | *1* |
| *Actual Output:* | *1* |

(e) *The first error state is when index* `i` *has the value* `0` *when it should have a value at the top of the array. Hence, the first error state is encountered immediately after the assignment to* `i` *in the for-statement if there is more than one value in* `x`.

| | |
|---|---|
| *Input:* | *x = [0, 1, 0]* |
| *Expected Output:* | *2* |
| *Actual Output:* | *0* |
| *First Error State:* | |
| | *x = [0, 1, 0]* |
| | *i = 0* |
| | *PC = just after* `i= 0;` |

(f) *See (a)*

`countPositive()` **Solution (Instructor only):**

(a) *The test in the conditional should be:*
`if (x[i] > 0) {`

(b) `x` *must be either null or empty. All other inputs result in the fault being executed. We give the empty case here.*

| | |
|---|---|
| *Input:* | *x = []* |
| *Expected Output:* | *0* |
| *Actual Output:* | *0* |

(c) *Any nonempty* `x` *without a 0 entry works fine.*

| | |
|---|---|
| *Input:* | *x = [1, 2, 3]* |
| *Expected Output:* | *3* |
| *Actual Output:* | *3* |

(d) *For this particular program, every input that results in error also results in failure. The reason is that error states are not repairable by subsequent processing. If there is a 0 in* `x`*, all subsequent states (after processing the 0) will be error states no matter what else is in* `x`*.*

(e)

| | |
|---|---|
| *Input:* | *x = [-4, 2, 0, 2]* |
| *Expected Output:* | *2* |
| *Actual Output:* | *3* |

*First Error State:*
*x = [-4, 2, 0, 2]*
*i = 2;*
*count = 2;*
*PC = immediately after the count++ statement.*

(f) *See (a)*

`oddOrPos()` **Solution:**

(a) *The if-test needs to take account of negative values (positive odd numbers are taken care of by the second test):*
`if (x[i]%2 == -1 || x[i] > 0)`

(b) `x` *must be either null or empty. All other inputs result in the fault being executed. We give the empty case here.*

| | |
|---|---|
| *Input:* | *x = []* |
| *Expected Output:* | *0* |
| *Actual Output:* | *0* |

(c) *Any nonempty* `x` *with only non-negative elements works, because the first part of the compound if-test is not necessary unless the value is negative.*

| | |
|---|---|
| *Input:* | *x = [1, 2, 3]* |
| *Expected Output:* | *2* |
| *Actual Output:* | *2* |

(d) *For this particular program, every input that results in error also results in failure. The reason is that error states are not repairable by subsequent processing. If there is a negative value in* `x`*, all subsequent states (after processing the negative value) will be error states no matter what else is in* `x`*.*

(e)

| | |
|---|---|
| *Input:* | *x = [-3, -2, 0, 1, 4]* |
| *Expected Output:* | *3* |
| *Actual Output:* | *2* |

*First Error State:*
*x = [-3, -2, 0, 1, 4]*
*i = 0;*
*count = 0;*
*PC = at end of* `if` *statement, instead of just before* `count++`

(f) *See (a)*