5. Below are four faulty programs. Each includes test inputs that result in failure. Answer the following questions about each program.

```
/**
 * Find last index of element
 *
 * @param x array to search
 * @param y value to look for
 * @return last index of y in x; -1 if absent
 * @throws NullPointerException if x is null
 */
public int findLast (int[] x, int y)
{
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
// test:  x = [2, 3, 5]; y = 2; Expected = 0
// Book website: FindLast.java
// Book website: FindLastTest.java
```

```
/**
 * Count positive elements
 *
 * @param x array to search
 * @return count of positive elements in x
 * @throws NullPointerException if x is null
 */
public int countPositive (int[] x)
{
   int count = 0;
   for (int i=0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}
// test:  x = [-4, 2, 0, 2]; Expcted = 2
// Book website: CountPositive.java
// Book website: CountPositiveTest.java
```

(a) Explain what is wrong with the given code. Describe the fault precisely by proposing a modification to the code.

(b) If possible, give a test case that does **not** execute the fault. If not, briefly explain why not.

(c) If possible, give a test case that executes the fault, but does **not** result in an error state. If not, briefly explain why not.

(d) If possible give a test case that results in an error state, but **not** a failure. Hint: Don't forget about the program counter. If not, briefly explain why not.

(e) For the given test case, describe the first error state. Be sure to describe the complete state.

(f) Implement your repair and verify that the given test now produces the expected output. Submit a screen printout or other evidence that your new program works.

Solutions:

===

findLast()

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

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

(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.*

(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.*

(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 or 1, depending on the compiler);*
	PC = just before `return -1;`*;*

(f) ···

===

countPositive()

(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.*

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

(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:	

$x = [-4, 2, 0, 2]$
$i = 2;$
$count = 1;$
PC = immediately before the count++ statement. (taking the branch to the count++ statement could be considered erroneous.)

(f) ···