EJERCICIO 1

1) Explica el fallo (qué está mal en el código). Describe de manera precisa el fallo y propón una modificación al código que lo arregle.

No he encontrado el error en este código.

2) Proporciona, si ello es posible, un caso de prueba que no ejecute el código que tiene el fallo. Si no es posible, explica por qué. Para cada caso de prueba indica los datos usados en la prueba, el resultado esperado y el resultado obtenido.

```
 /**
Test sin fallos
 */
@Test public void testelement(){
	int arg[] = {1, -2, 3};
	int obj = ejercicio_1.countPositive(arg);
	assertTrue("Numeros positivos", (obj == 2));
}
```

Los demás apartados son para cuando existe un fallo.

EJERCICIO 2

1) Explica el fallo (qué está mal en el código). Describe de manera precisa el fallo y propón una modificación al código que lo arregle.

Nos devuelve el primer cero en vez del último. Se modifica que en vez de que haya dentro del bucle un return, se cree una variable para que recorra todo el array y se pueda saber cual es el último.

```
public static int lastZero (int[] x){
	count = -1;
	 for (int i = 0; i < x.length; i++)
	 {
		if (x[i] == 0){
			count = i;
		}
	 }
return count;
}
```

2) Proporciona, si ello es posible, un caso de prueba que no ejecute el código que tiene el fallo. Si no es posible, explica por qué. Para cada caso de prueba indica los datos usados en la prueba, el resultado esperado y el resultado obtenido.

Que no haya ningún cero. Nunca entra en la condición del if, entonces no se ejecuta el return.

```
 /**
Test sin fallos
 */
@Test public void testlastZero(){
	int arg[] = {1, -2, 3};
	int obj = lastZero.lastZero(arg);
	assertTrue("Numeros positivos", (obj == -1));
}
```

3) Si es posible, proporciona un caso de prueba que ejecuta el fallo que hay en el código, pero que no provoque un error en el estado. Si no se puede, explica por qué.

Cuando solo hay un cero, ya que ejecutamos el error pero no lo podemos ver porque no se ha realizado bien el test, esta solución estaría bien.

```
 /**
Test sin fallos
 */
@Test public void testlastZero(){
        int arg[] = {1, -2, 0};
        int obj = lastZero.lastZero(arg);
        assertTrue("Numeros positivos", (obj == 2));
}
```

4) Si es posible, proporciona un caso de prueba que provoque un error en el estado, pero que no acabe provocando una disfunción en el comportamiento del programa. No olvides que el contador de programa forma parte, junto a las variables, del estado del programa. Si no es posible, explica por qué.

Al aparecer un cero más ya estaría mal realizado el programa.

```
 /**
Test con fallos
 */
@Test public void testlastZero_2(){
        int arg[] = {1, -2, 0, 0};
        int obj = lastZero.lastZero(arg);
        assertTrue("Numeros positivos", (obj == 3));
}
```

5) Para el caso de prueba del anterior apartado, describe el primero de los estados erróneos. Describe detalladamente todo el estado (todas las variables, incluyendo el contador de programa).

I= 0 → arg[i] = 1
i = 1 → arg[i] = -2
i = 2 → arg[i] = 0
return i
Devuelve la posición 2 en vez de la tres.

6**) Ejecuta en programas Java el código corregido y pruébalo con los casos de prueba creados en apartados anteriores.**
Corregido en el ej 1.

EJERCICIO 3

1) Explica el fallo (qué está mal en el código). Describe de manera precisa el fallo y propón una modificación al código que lo arregle.

No mira el primer elemento.

```
public static int findLast (int[] x, int y) {
        for (int i=x.length-1; i >= 0; i--)
        {
                if (x[i] == y)
                {
                        return i;
                }
        }
        return -1;
}
```

2) Proporciona, si ello es posible, un caso de prueba que no ejecute el código que tiene el fallo. Si no es posible, explica por qué. Para cada caso de prueba indica los datos usados en la prueba, el resultado esperado y el resultado obtenido.

Siempre se va a ejecutar el fallo, salvo que se reciba de parámetro un array NULL.

3) Si es posible, proporciona un caso de prueba que ejecuta el fallo que hay en el código, pero que no provoque un error en el estado. Si no se puede, explica por qué.

No provoca error porque aunque no mire el primer valor no está ahí justo el numero que queremos encontrar.

```
@Test public void testfindLast(){
        int arg[] = {1, -2, 0};
        int num = 0;
        int obj = findLast.findLast(arg, num);
        assertTrue("Numeros positivos", (obj == 2));
}
```

4) Si es posible, proporciona un caso de prueba que provoque un error en el estado, pero que no acabe provocando una disfunción en el comportamiento del programa. No olvides que el contador de programa forma parte, junto a las variables, del estado del programa. Si no es posible, explica por qué.

Se realiza con fallos porque no se encuetnra el tres que es el que estamos buscando porque no atraviesa el array esa posición para compararla.

```
/**
Test con fallos
 */
@Test public void testfindLast_1(){
        int arg[] = {3, -2, 0, 8, 10};
        int num = 3;
        int obj = findLast.findLast(arg, num);
        assertTrue("Numeros positivos", (obj == 0));
}
```

5) Para el caso de prueba del anterior apartado, describe el primero de los estados erróneos. Describe detalladamente todo el estado (todas las variables, incluyendo el contador de programa).

I = 1 → arg[i] = -2

Ya hay fallo porque no se ha encontrado el 3. Recorre el array hasta el final pero nunca pasa por la posición 0.

6) Ejecuta en programas Java el código corregido y pruébalo con los casos de prueba creados en apartados anteriores.

Corregido en el ej 1.

EJERCICIO 4

1) Explica el fallo (qué está mal en el código). Describe de manera precisa el fallo y propón una modificación al código que lo arregle.

No tiene en cuenta el cero como un numero positivo ni los números impares negativos.

```
public static int oddOrPos(int[] x){
        int count = 0;
        for (int i = 0; i < x.length; i++){
                if (x[i]%2 == 1 || x[i] >= 0 || x[i]%2 == -1)
                {
                        count++;
                }
        }
        return count;
}
```

2) Proporciona, si ello es posible, un caso de prueba que no ejecute el código que tiene el fallo. Si no es posible, explica por qué. Para cada caso de prueba indica los datos usados en la prueba, el resultado esperado y el resultado obtenido.

La única forma de que no se ejecutase el fallo es que salte la excepción NullPointerException y no se ejecute lo que hay dentro del bucle.

3) Si es posible, proporciona un caso de prueba que ejecuta el fallo que hay en el código, pero que no provoque un error en el estado. Si no se puede, explica por qué.

No salta la excepción porque tenemos 2 numeros positivos y no tiene que examinar la variable negativa.

```
@Test public void testOddOrPos(){
        int arg[] = {1, -2, 2};
        int obj = oddOrPos.oddOrPos(arg);
        assertTrue("Numeros positivos", (obj == 2));
}
```

4) Si es posible, proporciona un caso de prueba que provoque un error en el estado, pero que no acabe provocando una disfunción en el comportamiento del programa. No olvides que el contador de programa forma parte, junto a las variables, del estado del programa. Si no es posible, explica por qué.

Se realizan fallos porque el -3 también sería valido.

```
/**
	Test con fallos
	 */
	@Test public void testOddOrPos_1(){
		int arg[] = {1, -2, 2, -3};
		int obj = oddOrPos.oddOrPos(arg);
		assertTrue("Numeros positivos", (obj == 3));
	}
```

5) Para el caso de prueba del anterior apartado, describe el primero de los estados erróneos. Describe detalladamente todo el estado (todas las variables, incluyendo el contador de programa).

I= 0 → x[i] = 1 count = 1
i = 1 → x[i] = -2 count = 1
i = 2 → x[i] = 2 count = 2
I= 3 → x[i] = -3 count = 2. Tendría que ser count = 3.
x.length = 3 (se sale del bucle) devuelve 2.

6) Ejecuta en programas Java el código corregido y pruébalo con los casos de prueba creados en apartados anteriores.

Código en ejercicio 1.