README

Even Though the TA has explained the definition of an `error state', it remains somewhat confusing to me. Let's consider program 3 as an example, the `if` statement `if(x[i] >= 0)` is identified as a fault. If the condition is correctly satisfied/dissatisfied, does it result in an error state? I assume that it does result in error state because the internal state includes the outcome, ie. return value of `if` condition. Therefore, the fault can be executed without resulting in error state, and the all subsequent answers are based on this assumption.

Program 1

```
/**
* Find last index of element
*
* @param {Object[]} x - The array to search.
* @param {number} y - The value to look for.
*
* @returns {number} Last index of y in x; -1 if absent.
* @throws TypeError if x is not an array or y is not a
* number.
*/
function findLast(x, y) {
    if (!Array.isArray(x)) {
        throw new TypeError('The first parameter must be an array');
    }
    if (typeof y !== 'number') {
        throw new TypeError('The second parameter must be a number');
    }
    for (let i = x.length - 1; i > 0; i--) {
        if (x[i] === y) {
            return i;
        }
    }
    return -1;
}
// test: x = [2, 3, 5]; y = 2; Expected = 0
```

(a) Fault `i > 0` in the for loop `for (let i = x.length -1; i > 0; i--)`, it should be revised to `i >= 0` since the first element in array is `x[0]` and the iteration should end with `x[0]`.

(b) Test case `x = 3, y = 3; Expected = 0`. As `x` is not an array, the function encounters an error and halts execution. As a result, the fault is not executed.

(c) Test case `x = [3, 5, 2], y = 2; Expected = 2`. There is only one potential error state `(x = [3, 5, 2], y = 2, i = 0, PC = 'i > 0;', return value = false)`, as the return value of `i > 0` should be `true` so that the iteration can continue. However, with the given test case, the function returns the result during the first iteration and does not proceed to the final iteration, thereby avoiding the error state. As a result, the expected output and the actual output are both `2`.

(d) Test case `x = [1, 3, 5], y = 2; Expected = -1`. During the final iteration, there is an error state `(x = [1, 3, 5], y = 2, i = 0, PC = 'i > 0;', return value = false)`, as the return value of `i > 0` should be `true`. However, the error does not propagate to the output and result in a failure, because none of the elements in the array `x[]` are equal to `y`. As a result, the expected output and actual output are both `-1`.

(e) First error state is `(x = [1, 3, 5], y = 2, i = 0, PC = 'i > 0;', return value = false)` because the return value of `i > 0` should be `true`.

Program 2

```
/**
* Find last index of zero
*
* @param {Object[]} x - The array to search.
*
* @returns {number} Last index of 0 in x; -1 if absent.
* @throws TypeError if x is not an array.
*/
function lastZero(x) {
    if (!Array.isArray(x)) {
        throw new TypeError('Not an array');
    }
    for (let i = 0; i < x.length; i++) {
        if (x[i] === 0) {
            return i;
        }
    }
    return -1;
}
// test: x = [0, 1, 0]; Expected = 2
```

(a) Fault `for (let i = 0; i < x.length; i++)` iterate from the first to the last element in the array. However, for the purpose of this function, it is more appropriate to start the iteration from the last element. It is more appropriate to iterate backward `for (let i = x.length - 1; i >= 0; i++)`.

(b) Test case `x = 0; Expected = 0`. As `x` is not an array, the function encounters an error and halts execution. As a result, the fault is not executed.

(c) It's not possible to execute the fault without resulting in error state. The initial setting of `i = 0` instead of `i = x.length - 1` not only execute the fault but also simultaneously leads to error state e.g., `(x = [1, 1, 0], i = 0, PC = 'let i = 0')`, as `i` should be `x.length - 1`.

(d) Test case `x = [1, 1, 1], Expected = -1`. The function execute the fault and result in error state `(x = [1, 1, 1], i = 0, PC = 'let i = 0;')`, as `i` should be `x.length - 1`. However, there is no zero in present in the array and the function returns the correct value `-1`, resulting in no failure. As a result, the expected output and actual output are both `-1`.

(e) First error state is `(x = [1, 1, 1], i = 0, PC = 'let i = 0;')`.

Program 3

```
/**
* Count positive elements
*
* @param {Object[]} x - The array to search.
*
* @returns {number} Count of positive elements in x.
* @throws TypeError if x is not an array.
*/
    function countPositive(x) {
        if (!Array.isArray(x)) {
            throw new TypeError('Not an array');
        }
    let count = 0;
    for (let i = 0; i < x.length; i++) {
        if (x[i] >= 0) {
            count++;
        }
    }
    return count;
}
// test: x = [-4, 2, 0, 2]; Expcted = 2
```

(a) Fault `if(x[i] >= 0)`. The `if` condition is satisfied if an element in the array is equal to zero, resulting in fault because the purpose of the function is to count the number of positive numbers. The code should be revised to `if(x[i] > 0)`.

(b) Test case `x = 1; Expected = 1` does not execute fault since `x` is not an array and it halts execution of the function.

(c) Test case `x = [-4, 2, -1, 2]; Expected = 2`. Despite executing the fault for each iteration of the for loop, none of the elements in the array equal to zero. As a result, `if(x[i] >= 0)` doesn't incorrectly return `true` and `count` is not incorrectly accumulated, thereby result in no error state. As a result, the expected output and actual output are both `2`.

(d) It's not possible to give a test case that result in error state without leading to a failure. An error state `(x = [-4, 2, 0, 2], count = 1, i = 2, PC = 'if (x[i] >= 0)', return value = true)` for example incorrectly returns `true` and lets `count` be accumulated in the next statement `count++;`. Therefore the error state propagate to the output, the value of `count` can't be coincidentally correct, resulting in failure.

(e) There's no test case in (d). For test case in the example, the first error state is `(x = [-4, 2, 0, 2], count = 1, i = 2, PC = 'if (x[i] >= 0)', return value = true)` since the return value should be `false`.

Program 4

```
/**
* Count odd or postive elements
*
* @param {Object[]} x - The array to search.
*
* @return {number} Count of odd/positive values in x.
* @throws TypeError if x is not an array.
*/
function oddOrPos(x) {
    if (!Array.isArray(x)) {
        throw new TypeError('Not an array');
    }
    let count = 0;
    for (let i = 0; i < x.length; i++) {
        if (x[i] % 2 === 1 || x[i] > 0) {
            count++;
        }
    }
    return count;
}
// test: x = [-3, -2, 0, 1, 4]; Expected = 3
```

(a) Fault `if (x[i] % 2 === 1 || x[i] > 0)` prevent odd negative elements from being counted during the iteration since if condition both `x[i] % 2 === 1` and `x[i] > 0` are not satisfyied. Therefore, the code should be revised to `if( x[i] % 2 === 1 || x[i] % 2 === -1 || x[i] > 0)`.

(b) Test case `x = 1; Expected = 1` does not execute fault because `x` is not an array. Instead it triggers an error and halts the execution of function, preventing the fault from executing.

(c) Test case `x = [3, -2, 0, 1, 4]; Expected = 3` executes the fault. However, there are no odd negative elements that incorrectly dissatisfy the `if` condition `if (x[i] % 2 === 1 || x[i] > 0)` and make this fault returns `false`, the test case doesn't result in error state.

(d) It's not possible to result in error state without also resulting in failure of the function. Error states (e.g., `(x = [-3, -2, 0, 1, 4], i = 0, count = 0, PC = 'if (x[i] % 2 === 1 || x[i] > 0)', return value = false))`) lead to incorrect control flow of the function and cause the execution of `count++;`, and they are propagated to the output and result in failure without exception.

(e) There's no test case in (d). For test case in the example, the first error state is `(x = [-3, -2, 0, 1, 4], i = 0, count = 0, PC = 'if (x[i] % 2 === 1 || x[i] > 0)', return value = false))` since the return value should be `true`.