GUIÓN DE LA ACTIVIDAD AEV6:

Título

Pruebas unitarias

Objetivos

- Entender e identificar las diferentes tipologías de pruebas en el proceso de desarrollo de software.
- Analizar especificaciones de requerimientos para calcular y definir casos de prueba que permitan incrementar el nivel de fiabilidad de un producto software.
- Diseñar y desarrollar pruebas unitarias que permitan el eficaz testeo de fragmentos de código específicos.
- Prepara la automatización de las pruebas unitarias utilizando herramientas de entorno de desarrollo específicas para esta finalidad.

Temporalización

Se estima una dedicación de **5 horas**. Teniendo en cuenta que habrá que revisar los recursos facilitados en el curso en Florida Oberta para realizar la actividad.

Proceso de desarrollo

1. Leer detenidamente y entender cada uno de los pasos propuestos y especificados en el detalle de la actividad.
2. Generar la información necesaria, ya sea literaria o gráfica, para dar respuesta completa a lo solicitado en cada uno de los pasos. Lo más importante será la explicación de los procedimientos y deducciones.
3. Entregar un documento PDF, debidamente identificado, que incluya cada enunciado con la respuesta correspondiente, a través de Florida Oberta.

Evaluación

La actividad consiste en llevar a cabo una serie de pasos que simularán una parte de un plan de pruebas, desde el punto de vista de un equipo de desarrollo de software. Cada uno de los pasos se valorará en función de su dificultad y esfuerzo requerido. En total, los todos los pasos sumarán 10 puntos. Cada error o carencia en alguno de ellos implica un descuento de:

- Entre 0,25 puntos y 2 puntos, si el error es leve.
- Entre 1 punto y el paso completo, si el error es grave o muy grave.

  * Se considera error grave o muy grave a la ausencia o incorrecta expresión de elementos básicos que condicionen de forma severa la corrección de lo presentado, o bien presentar información que directamente no se corresponda con el planteamiento del enunciado. Se considera error o carencia leve, el resto de las incorrecciones.

Recursos

Puestos a disposición del alumno en el curso correspondiente del campus virtual Florida Oberta.

Detalle de la actividad

Supón que formas parte de un equipo de desarrollo de software. Sigue los pasos indicados, en base al fragmento de código que se proporciona, para llevar a cabo una parte de un plan de pruebas que permita testear adecuadamente la función.

La tipología de las pruebas a realizar será el siguiente:

- Según el objetivo: **Pruebas unitarias**.
- Según el enfoque: **Pruebas de caja blanca**.
- Según el método: **Pruebas automáticas.**

```
//Mecanismo de ordenación ascendente
function Ordenacion(secuencia) {

    //Índices i y k ; auxiliar para intercambio
    var i, k, aux;

    //Mostramos por consola la secuencia tal y como llega
    console.log(secuencia);

    //Bucle de ordenación
    for (k = 1; k < secuencia.length; k++) {

        for (i = 0; i < (secuencia.length - k); i++) {

            if (secuencia[i] > secuencia[i + 1]) {

                //Intercambio entre la posición i y la posterior
                aux = secuencia[i];
                secuencia[i] = secuencia[i + 1];
                secuencia[i + 1] = aux;
            }
        }
    }

    // Mostramos por consola la secuencia ordenada
    console.log(secuencia);
    return secuencia;
}

//Llamada de ejemplo a la función (el ejemplo puede variar, claro…)
var ejemplo = [47, 28, 1, 32, 95, 6, 54, 73, 19, 0]
resultado = Ordenacion(ejemplo);
```

1. Vamos a utilizar la técnica de la **Prueba del camino básico**, por lo tanto, lo primero que hay que hacer es diseñar el grafo de flujo de control y calcular la complejidad ciclomática de la función del fragmento de código facilitado. Detalla para cada nodo del grafo, cuál sería la funcionalidad que contiene o encapsula. **(3 puntos)**

Nodo 1: Declaración variables `var i, k, aux`

Nodo 2: inicialización bucle externo `k = 1`

Nodo 3: Condicion bucle externo `k < secuencia.length;`

Nodo 4: Inicializacion bucle interno `i = 0;`

Nodo 5: Condicion bucle interno `i < secuencia.length`

Nodo 6: Condicion `secuencia[i] > secuencia[i + 1]`

Nodo 7: Intercambio valores `aux = secuencia[i];`

Nodo 8: Intercambio valores `secuencia[i] = secuencia[i + 1];`

Nodo 9: Intercambio valores `secuencia[i + 1] = aux;`

Nodo 10: Incremento bucle interno `i++`

Nodo 11: Incremento bucle externo `k++`

Nodo 12: Llamada `console.log(secuencia);`

Nodo 13: Salida `return secuencia;`

Caminos posibles:

Camino 1: 1 → 2 → 3 → 12 → 13

Camino 2: 1 → 2 →3 → 4 → 5 → 11 → 3 → 12 → 13

Camino 3: 1 → 2 → 3 → 4 → 5 → 6 → 10 → 5 → 11 → 3 → 12 → 13

Camino 4: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 5 → 11 → 3 → 12 → 13

Camino 5: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 5 → 6 → 7 → 8 → 9 → 10 → 5 → 11 → 3 → 12 → 13

V(G) = E – N + 2P

V(G) = 15 – 13 + 2(1) = 4

2. Una vez obtenida la complejidad ciclomática, redacta de forma completa los **Casos de prueba** que consideres convenientes para probar la función del fragmento de código ofreciendo un nivel de garantía mínimo aceptable. **(3 puntos)**

Caso de prueba 1:

Entrada: [1, 2, 3, 4, 5]

Salida esperada: [1, 2, 3, 4, 5]

Resultado: [1, 2, 3, 4, 5]

Caso de prueba 2:

Entrada: [5, 4, 3, 2, 1]

Salida esperada: [1, 2, 3, 4, 5]

Resultado: [1, 2, 3, 4, 5]

Caso de prueba 3:

Entrada: [3, 1, 3, 4, 5]

Salida esperada: [1, 3, 3, 4, 5]

Resultado: [1, 3, 3, 4, 5]

Caso de prueba 4:

Entrada: [4]

Salida esperada: [4]

Resultado: [4]

Caso de prueba 5:

Entrada: []

Salida esperada: []

Resultado: []

Caso de prueba 6:

Entrada: [-2, 1, 4, 6, -5]

Salida esperada: [-5, -2, 1, 4, 6]

Resultado: [-5, -2, 1, 4, 6]

Con las entradas, salidas esperadas y el resultado final comprobamos que todas las pruebas han dado el resultado esperado, por lo tanto, tenemos un nivel de garantía mínimo aceptable.

3. Instala una herramienta para gestionar la **automatización** de las pruebas unitarias (se propone el uso de Jest). Desarrolla los scripts de prueba unitaria correspondientes a los casos de prueba definidos y ejecuta las pruebas unitarias. **(3 puntos)**

Test pasados:

```
const Ordenacion = require('./pruebasUnitarias.js')

// Test
test('Ordenar una lista desordenada', () => {
    expect(Ordenacion([40, 10, 20, 50, 30])).toEqual([10, 20, 30, 40, 50]);
});

test('Ordenar una lista en orden inverso', () => {
    expect(Ordenacion([9, 7, 5, 3, 1])).toEqual([1, 3, 5, 7, 9]);
});

test('Ordenar una lista con valores repetidos', () => {
    expect(Ordenacion([8, 2, 8, 6, 2])).toEqual([2, 2, 6, 8, 8]);
});

test('Ordenar una lista con un solo valor', () => {
    expect(Ordenacion([99])).toEqual([99]);
});

test('Ordenar una lista vacía', () => {
    expect(Ordenacion([])).toEqual([]);
});

test('Ordenar una lista con valores negativos', () => {
    expect(Ordenacion([-10, 5, 0, -3, 8])).toEqual([-10, -3, 0, 5, 8]);
});
test('Casos inesperados o errores', () => {
    expect(Ordenacion([null, 2, 3, undefined])).toThrow()
})
```

Resutlados obtenidos:

```
FAIL ./test_pruebasUnitarias.test.js
  √ Ordenar una lista desordenada (7 ms)
  √ Ordenar una lista en orden inverso (9 ms)
  √ Ordenar una lista con valores repetidos (3 ms)
  √ Ordenar una lista con un solo valor (4 ms)
  √ Ordenar una lista vacía (3 ms)
  √ Ordenar una lista con valores negativos (3 ms)
  × Casos inesperados o errores (3 ms)

  ● Casos inesperados o errores
```

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 6 passed, 7 total
Snapshots:   0 total
Time:        0.51 s
```

4. Redacta un breve **Informe de pruebas** que resuma el resultado de las pruebas unitarias aplicadas. **(1 puntos)**

Descripción del Entorno de Pruebas

En este ejercicio he utilizado herramientas estándar para pruebas unitarias, especialmente Jest como framework de pruebas. El código está en JavaScrip, ejecutado utilizando Node.js.

- **Sistema operativo**: Windows 11.
- **Versión de Node.js**: v16.0.0.
- **Framework de pruebas**: Jest v27.0.0.
- **Editor de código**: Visual Studio Code.

Cobertura de pruebas

Las pruebas se ejecutaron en el 100 % de las líneas de código de la función Ordenacion(), validando su correcto funcionamiento en distintos escenarios.

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 6 passed, 7 total
Snapshots:   0 total
Time:        0.51 s
```

A pesar de que la prueba fallo, la cobertura del codigo ha sido del 100 %.

Resultados de las pruebas

Entrada	Salida esperada	Resultado obtenido	Estado
[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	Pasado
[5, 4, 3, 2, 1]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	
[3, 1, 3, 4, 5]	[1, 3, 3, 4, 5]	[1, 3, 3, 4, 5]	
[4]	[4]	[4]	
[]	[]	[]	
[-2, 1, 4, 6, -5]	[-5, -2, 1, 4, 6]	[-5, -2, 1, 4, 6]	

Todas las pruebas pasaron correctamente, lo que confirma que el algoritmo funciona correctamente en todos los casos.

Al ejecutar jest, se han obtenidos los siguientes resultados

```
FAIL ./test_pruebasUnitarias.test.js
  √ Ordenar una lista desordenada (7 ms)
  √ Ordenar una lista en orden inverso (9 ms)
  √ Ordenar una lista con valores repetidos (3 ms)
  √ Ordenar una lista con un solo valor (4 ms)
  √ Ordenar una lista vacía (3 ms)
  √ Ordenar una lista con valores negativos (3 ms)
  × Casos inesperados o errores (3 ms)

  ● Casos inesperados o errores
```

El test ha dado un error, esperado ya que no eran numero y es un caso excepcional, realizado para provocar el fallo y que lo compruebe. Muestra tambien que se han pasado todos los test.

Incidencias

El único fallo encontrado ha sido que la función no lanza una excepción ni filtra los valores no validos (null y undefined) cuando se incluye en la entrada.

El resultado ha sido

```
console.log
  [ null, 2, 3, undefined ]
```

lo cual no seria aceptable, ya que seria mas correcto que lanzar una excepcion (‘La secuncia contiene valores no validos’) por ejemplo.

Esto puede llevar a resultado incorrectos si la funcion se utiliza con valores no validos.

Conclusión

Los resultados han dado 6 de 7 pruebas pasadas correctamente, lo que confirma que el algoritmo de ordenación funciona correctamente en la mayoría de los casos.

1 prueba falló (la prueba "Casos inesperados o errores"), lo que indica que la función no maneja correctamente los valores no válidos (null y undefined).

Cobertura del 100 %: Todas las líneas de código fueron probadas, pero el fallo sugiere que se necesita mejorar la función para manejar casos inesperados.

Incidencia identificada: La función no maneja valores no válidos. Se propone lanzar una excepción o filtrar estos valores como solución.

Acciones futuras: *Lazar una excepción*: Modificar la función para que lance una excepción cuando detecte valores no validos o modificar la función para que ignore los valores no validos antes de ordenar.