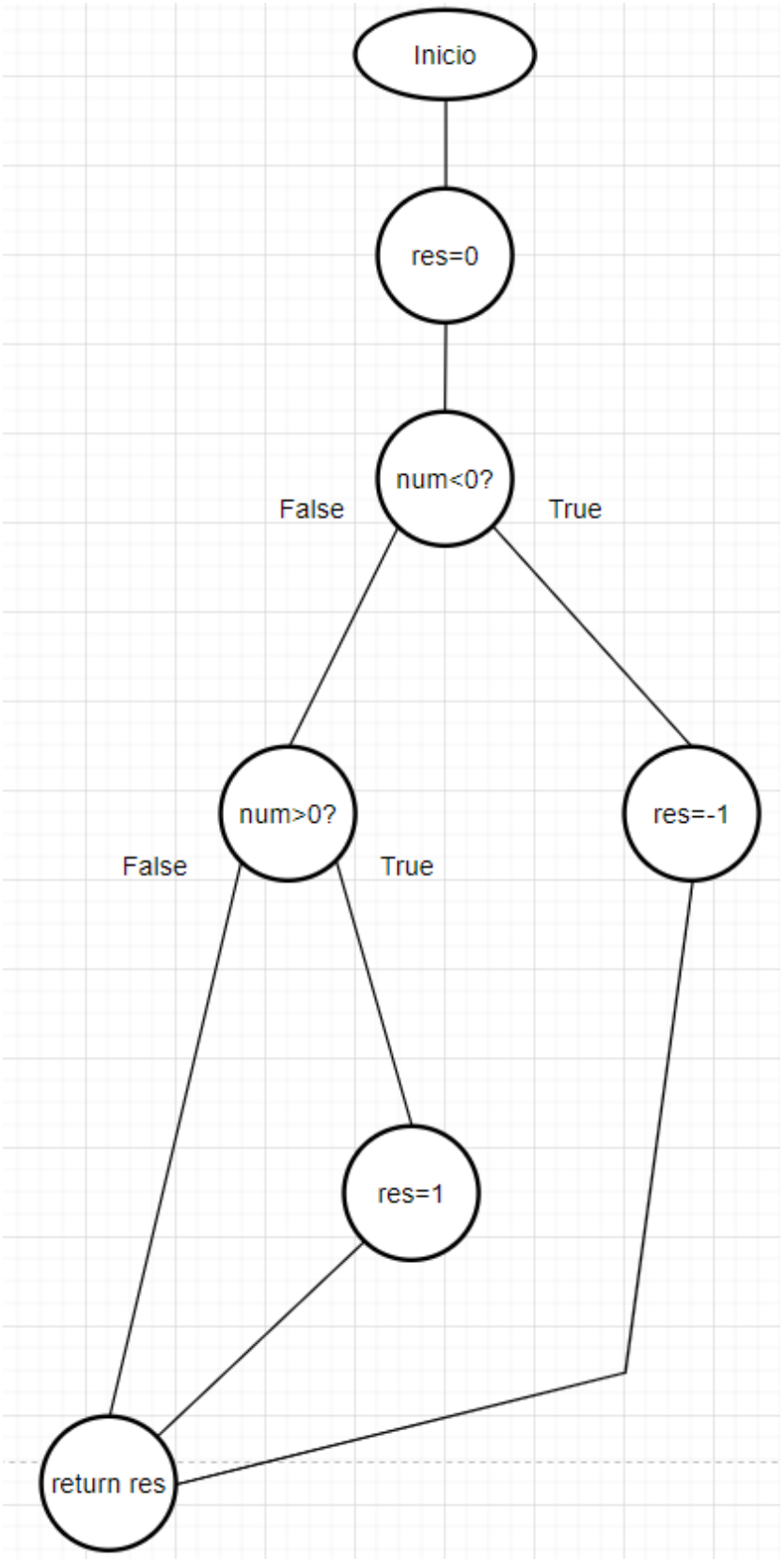
Ejercicio 1

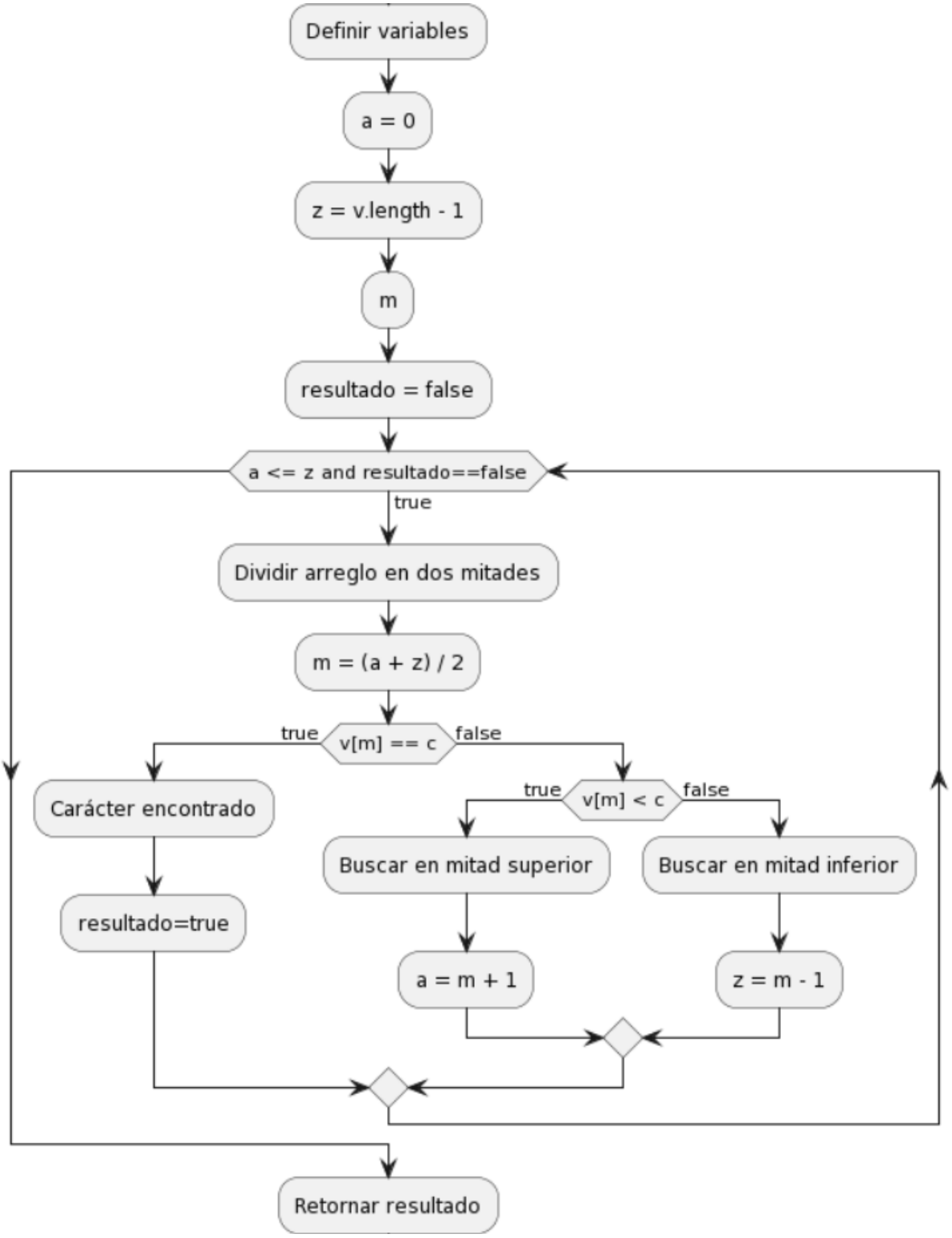


El grafo de flujo tiene una estructura lineal, por lo que solo hay un camino posible a través del programa. Por lo tanto, la complejidad de McCabe de este código es 1. Esto significa que no hay estructuras de control complejas en el código y que es relativamente simple de entender y mantener.

Ejercicio 2

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class FacturaTest {

    @Test
    public void MenorOIgualA300() {
        Factura factura = new Factura();
        factura.setConsumokWh(250);
        assertEquals(9, factura.calcularPrezokWh());
    }

    @Test
    public void MayorQue300MenorOIgualA600() {
        Factura factura = new Factura();
        factura.setConsumokWh(500);
        assertEquals(8, factura.calcularPrezokWh());
    }

    @Test
    public void MayorQue600MenorOIgualA1000() {
        Factura factura = new Factura();
        factura.setConsumokWh(800);
        assertEquals(6, factura.calcularPrezokWh());
    }

    @Test
    public void MayorQue1000MenorOIgualA2000() {
        Factura factura = new Factura();
        factura.setConsumokWh(1500);
        assertEquals(5, factura.calcularPrezokWh());
    }

    @Test
    public void MayorQue2000() {
        Factura factura = new Factura();
        factura.setConsumokWh(3000);
        assertEquals(0, factura.calcularPrezokWh());
    }
}
```

Ejercicio 3


Definir variables
a = 0
z = v.length - 1
m
resultado = false
a <= z and resultado==false
true
Dividir arreglo en dos mitades
m = (a + z) / 2
true
v[m] == c
false
Carácter encontrado
resultado=true
true
v[m] < c
false
Buscar en mitad superior
Buscar en mitad inferior
a = m + 1
z = m - 1
Retornar resultado

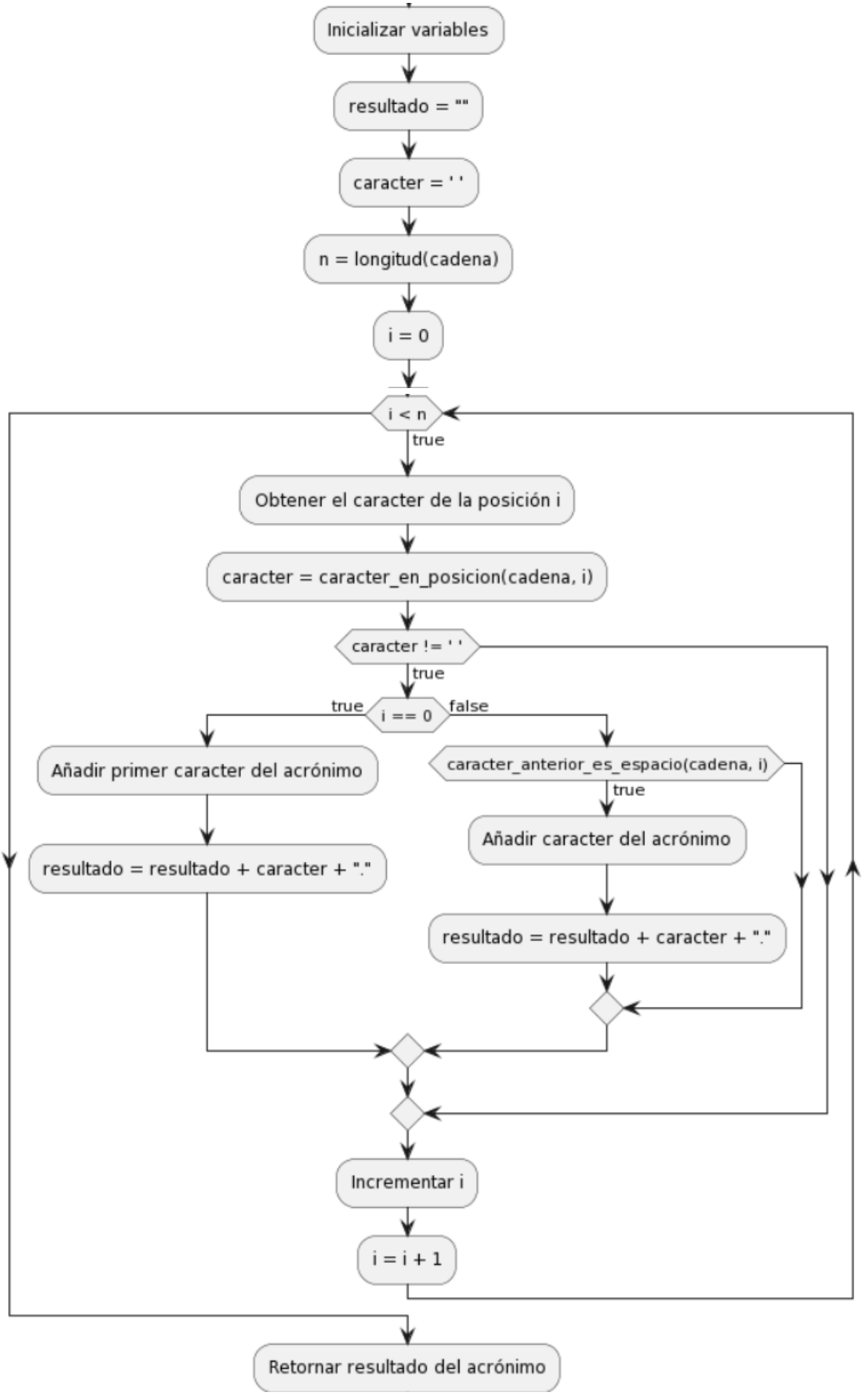
La complejidad de McCabe para este código es 3. Hay un solo punto de entrada (el método busca), un solo punto de salida (la sentencia return), y tres posibles caminos en la estructura de control de flujo:

1-Si la condición if (v[m] == c) se cumple, el flujo continúa con la sentencia resultado=true, saliendo del ciclo while en la siguiente iteración.

2-Si la condición if (v[m] < c) se cumple, el flujo continúa con la sentencia a = m + 1, llevando a la siguiente iteración del ciclo while.

3-Si ninguna de las dos condiciones anteriores se cumple, el flujo continúa con la sentencia z = m - 1, llevando a la siguiente iteración del ciclo while.

Ejercicio 4


Inicializar variables
resultado = ""
caracter = ' '
n = longitud(cadena)
i = 0
i < n
true
Obtener el caracter de la posición i
caracter = caracter_en_posicion(cadena, i)
caracter != ' '
true
true
i == 0
false
Añadir primer caracter del acrónimo
resultado = resultado + caracter + "."
caracter_anterior_es_espacio(cadena, i)
true
Añadir caracter del acrónimo
resultado = resultado + caracter + "."
Incrementar i
i = i + 1
Retornar resultado del acrónimo

El código tiene una complejidad de McCabe de 3, lo que significa que hay 3 caminos distintos a través del código los cuales son:

1-Flujo principal (resultado = cadena vacía): este camino se sigue si la cadena de entrada es una cadena vacía. En este caso, el resultado también es una cadena vacía.

2-Camino donde el primer carácter de la cadena es una letra (resultado = carácter + '.'): este camino se sigue si el primer carácter de la cadena no es un espacio en blanco. En este caso, el primer carácter de la cadena se añade al resultado, seguido de un punto.

3-Camino donde un carácter en la cadena es una letra que sigue a un espacio en blanco (resultado = resultado + carácter + '.'): este camino se sigue si el carácter actual de la cadena no es un espacio en blanco y el carácter anterior en la cadena es un espacio en blanco. En este caso, el carácter actual de la cadena se añade al resultado, seguido de un punto.

Ejercicio 5

```java
public class Calculadora3 {
    public int suma(int a, int b) {
        return a + b;
    }
    public int resta(int a, int b) {
        return a - b;
    }
    public int multiplicacion(int a, int b) {
        return a * b;
    }
    public int division(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException(s: "No se puede dividir por cero");
        }
        return a / b;
    }
}

```

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class Calculadora3Test {

    @Test
    public void testSuma() {
        Calculadora3 calc = new Calculadora3();
        int resultado = calc.suma(a: 3, b: 4);
        assertEquals(7, resultado);
    }

    @Test
    public void testResta() {
        Calculadora3 calc = new Calculadora3();
        int resultado = calc.resta(a: 5, b: 2);
        assertEquals(3, resultado);
    }

    @Test
    public void testMultiplicacion() {
        Calculadora3 calc = new Calculadora3();
        int resultado = calc.multiplicacion(a: 6, b: 4);
        assertEquals(24, resultado);
    }

    @Test
    public void testDivision() {
        Calculadora3 calc = new Calculadora3();
        int resultado = calc.division(a: 10, b: 2);
        assertEquals(5, resultado);
    }
}
```