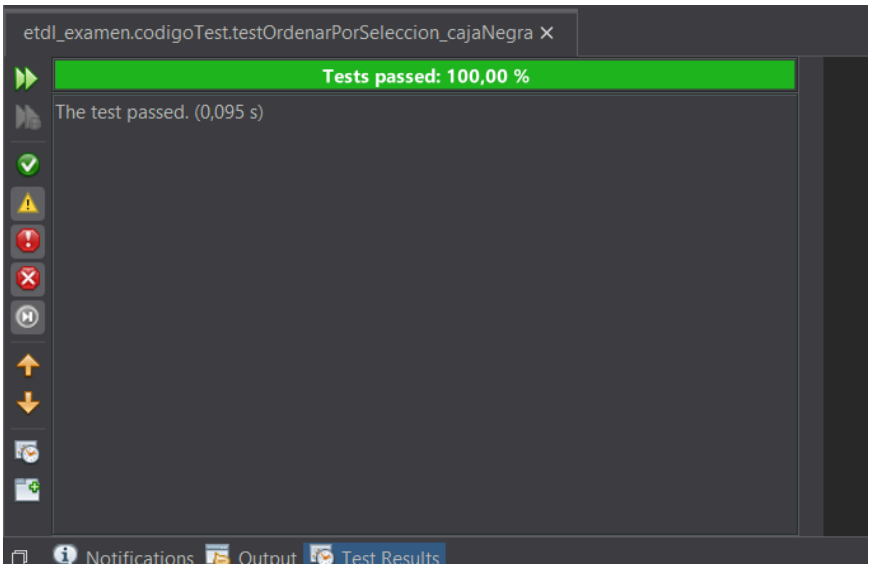
```java
public class codigo {

    public static boolean ordenarPorSeleccion(int[] listaNumeros) {

        int temporal;
        boolean listaOrdenada = true;

        for(int i = 0; i < listaNumeros.length - 1; i++) {
            for(int j = i + 1; j < listaNumeros.length; j++) {
                if(listaNumeros[i] > listaNumeros[j]) {
                    temporal = listaNumeros[i];
                    listaNumeros[i] = listaNumeros[j];
                    listaNumeros[j] = temporal;

                    listaOrdenada = false;
                }
            }
        }

        return listaOrdenada;
    }

}
```

Ejercicio 1

Realizamos las pruebas de caja negra

```java
@Test
public void testOrdenarPorSeleccion_cajaNegra() {
    assertTrue(condition:codigo.ordenarPorSeleccion(new int[]{1,2,3,4,5,6,7,8,9,10}));
    assertEquals(expected: false, actual:codigo.ordenarPorSeleccion(new int[]{1,3,2,5,4,7,6,9,8,10}));
}
```

Para asegurarnos de que el código funciona como debe de funcionar, le pasamos un vector de números ordenados y otro desordenado. Si es correcto, debe pasar ambas pruebas ya que devuelve true si el vector no esta ordenado (como en el primer caso), y devolverá false si el vector tenía que ser ordenado (como en el segundo caso)

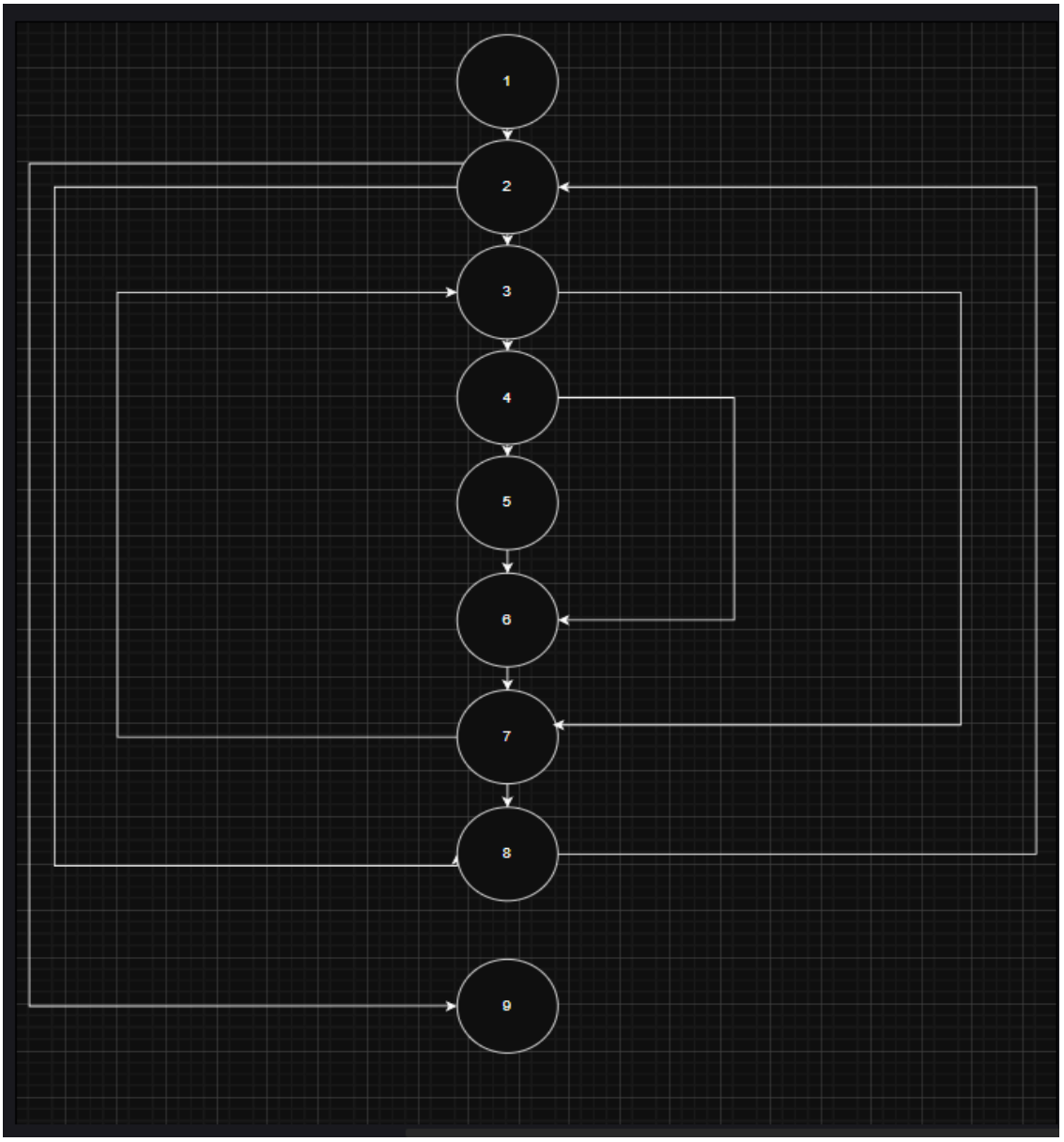


Ejercicio 2

Primero, procedemos a dividir el código en instrucciones y nodos

```java
public static boolean ordenarPorSeleccion(int[] listaNumeros) {

    int temporal;                                                   // instruccion 1 - nodo 1
    boolean listaOrdenada = true;                                   // instruccion 2

    for(int i = 0; i < listaNumeros.length - 1; i++) {              // instruccion 3 - nodo 2
        for(int j = i + 1; j < listaNumeros.length; j++) {          // Instruccion 4 - nodo 3
            if(listaNumeros[i] > listaNumeros[j]) {                 // instruccion 5 - nodo 4
                temporal = listaNumeros[i];                         // instruccion 6 - nodo 5
                listaNumeros[i] = listaNumeros[j];                  // instruccion 7
                listaNumeros[j] = temporal;                         // instruccion 8

                listaOrdenada = false;                              // instruccion 9
            }                                                       // fin del if - instruccion 10 - nodo 6
        }                                                           // fin del for j - instruccion 11 - nodo 7
    }                                                               // fin del for i - instruccion 12 - nodo 8

    return listaOrdenada;                                           // instruccion 12 - nodo 9
}
```

a)



b)

ARISTAS - NODOS + 2

11 - 9 + 2 = 4

DECISIONES + 1

- for(int i = 0; i < listaNumeros.length - 1; i++)
- for(int j = i + 1; j < listaNumeros.length; j++)
- if(listaNumeros[i] > listaNumeros[j])

3 + 1 = 4

REGIONES CERRADAS + 1

- 2-8
- 3-7
- 4-6

3 + 1 = 4

c)

Como la complejidad ciclomática es 4, significa que hay 4 caminos posibles

1 - 2 - 8 - 2 - 9 .

1 - 2 - 3 - 7 - 8 - 2 - 9

1 - 2 - 3 - 4 - 6 - 7 - 8 - 2 - 9

1 - 2 - 3 - 7 - 8 - 2 - 9 .

Ejercicio 3

Procedemos a realizar las pruebas de caja blanca para cubrir todos los caminos posibles

```
@Test
public void testOrdenarPorSeleccion_cajaBlanca() {
    assertTrue(condition:codigo.ordenarPorSeleccion(new int[]{1,2,3,4,5,6,7,8,9,10})); // 1 - 2 - 3 - 7 - 8 - 2 - 9 (Lista ordenada)
    assertTrue(condition:codigo.ordenarPorSeleccion(new int[]{}));                     // 1 - 2 - 8 - 2 - 9 (Lista vacia)
    assertTrue(condition:codigo.ordenarPorSeleccion(new int[]{1,2}));                  // 1 - 2 - 3 - 7 - 8 - 2 - 9 (Lista de 2 elementos ordenad
    assertTrue(condition:codigo.ordenarPorSeleccion(new int[]{0,1}));                  // 1 - 2 - 3 - 4 - 6 - 7 - 8 - 2 - 9 (Lista de 2 elementos
}
```