

# UT05. DISEÑO Y REALIZACIÓN DE PRUEBAS

Entornos de Desarrollo

1 DAW – C.I.F.P. Carlos III - Cartagena

# Índice

---

## **1.- Planificación de las pruebas.**

## **2.- Tipos de prueba.**

2.1.- Funcionales.

2.2.- Estructurales.

2.3.- Regresión.

## **3.- Procedimientos y casos de prueba.**

## **4.- Herramientas de depuración.**

4.1.- Puntos de ruptura.

4.2.- Tipos de ejecución.

4.3.- Examinadores de variables.

## **5.- Validaciones.**

## **6.- Pruebas de código.**

6.1.- Cubrimiento.

6.2.- Valores límite.

6.3.- Clases de equivalencia.

## **7.- Normas de calidad.**

## **8.- Pruebas unitarias.**

8.1.- Herramientas para Java.

8.2.- Herramientas para otros lenguajes.

## **9.- Automatización de la prueba.**

## **10.- Documentación de la prueba.**

# Funcionales. Análisis de Valores Límite

## ▪ Análisis de Valores Límite

- Complementa las particiones de equivalencia, y ejercitan los valores justo por encima y por debajo de los márgenes de la clase de equivalencia.
- Se basa en que los errores tienden a producirse con más probabilidad en los límites o extremos de los campos de entrada
- Reglas
  - Si hay rango de valores → casos válidos para los extremos y no válidos para los valores más allá de los extremos.
  - Si hay número de valores → valores máximo y mínimo, un valor por encima de máximo y otro por el mínimo
  - Aplicar las reglas 1 y 2 para las condiciones de salida
  - Aplicar reglas para estructuras tipo array, con límites establecidos
- Ejemplo.
  - Función: `esMayorDeEdad (persona)`
  - Valores límite: `-1, 0, 17, 18`

# Ejemplo

	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	
Puesto	Alfanumérico de hasta 4 caracteres	
Antigüedad	De 0 a 25 años (Real)	
Horas semanales	De 0 a 60	
Fichero de entrada	Tiene de 1 a 100 registros	
Fichero de salida	Podrá tener de 0 a 10 registros	
Array interno	De 20 cadenas de caracteres	

# Ejemplo

	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	Valores: 0, 1, 100, 101
Puesto	Alfanumérico de hasta 4 caracteres	Longitud de caracteres: 0, 1, 4, 5
Antigüedad	De 0 a 25 años (Real)	Valores: 0, 25, -0.1, 25.1
Horas semanales	De 0 a 60	Valores: 0, 60, -1, 61
Fichero de entrada	Tiene de 1 a 100 registros	Para leer 0,1, 100, 101
Fichero de salida	Podrá tener de 0 a 10 registros	Para generar 0, 10, y 11 registros (ni se puede generar -1 registro)
Array interno	De 20 cadenas de caracteres	Para el primer y último elemento

# Ejemplos

- Partimos del empleado (que tiene que ser un número de tres dígitos que no empiece por 0) del ejemplo 1. Utilizando esta técnica, para la clase de equivalencia V1 que representa un rango de valores ( $100 \geq \text{Empleada} \leq 999$ ) se debe generar dos casos de prueba con el límite inferior y el superior del rango

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Empleado	Departamento	Oficio	
CP13	(1) (3) (4)	100	20	Programador	S3
CP14	(1) (2) (5)	999		Analista	S1
CP15	(8) (3) (6)	99	30	Diseñador	ER1
CP16	(9) (2) (4)	1000		Programador	ER1

# Funcionales. Pruebas aleatorias

---

- **Pruebas aleatorias**

- Consiste en generar entradas aleatorias para la aplicación que hay que probar.
- Se suelen utilizar generadores de prueba, capaces de crear un volumen de casos de prueba al azar.
- Se suelen utilizar en aplicaciones no interactivas.

# Estructurales

---

- Conjunto de pruebas de la **caja blanca**
- **Función:** comprobar
  - que se van a ejecutar todas la instrucciones del programa,
  - no hay código no usado,
  - que los caminos lógicos del programa se van a recorrer, etc.



# Estructurales. Criterios de cobertura

- Criterios de cobertura
  - **Cobertura de sentencias**
    - Generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.
  - **Cobertura de decisiones**
    - Crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez a cierto y otra a falso.
  - **Cobertura de condiciones**
    - Cada condición de una decisión se evalúa al menos una vez a falso y otra a verdadero.

Decisión = condición1 + condición2

```
if (numero>0 && repetir==true)
```

- Condiciones:
  - num>0
  - repetir==true
- Decisiones
  - if (true or false)

## Estructurales. Criterios de cobertura (cont)

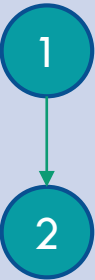
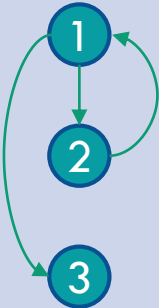
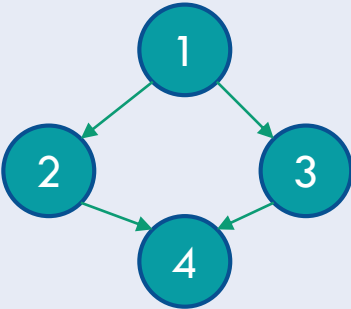

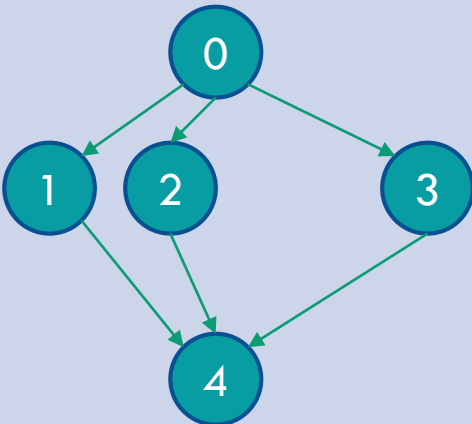
---

- **Cobertura de condiciones y decisiones**
  - Consiste en cumplir simultáneamente las dos anteriores.
- **Cobertura de caminos**
  - Ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final → **camino**.
  - Reducción del número de caminos → **camino prueba**.
- **Cobertura del camino de prueba: dos opciones**
  - Cada bucle se debe ejecutar sólo una vez
  - Cada bucle se ejecuta tres veces: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces.

# Estructurales. Prueba de camino básico

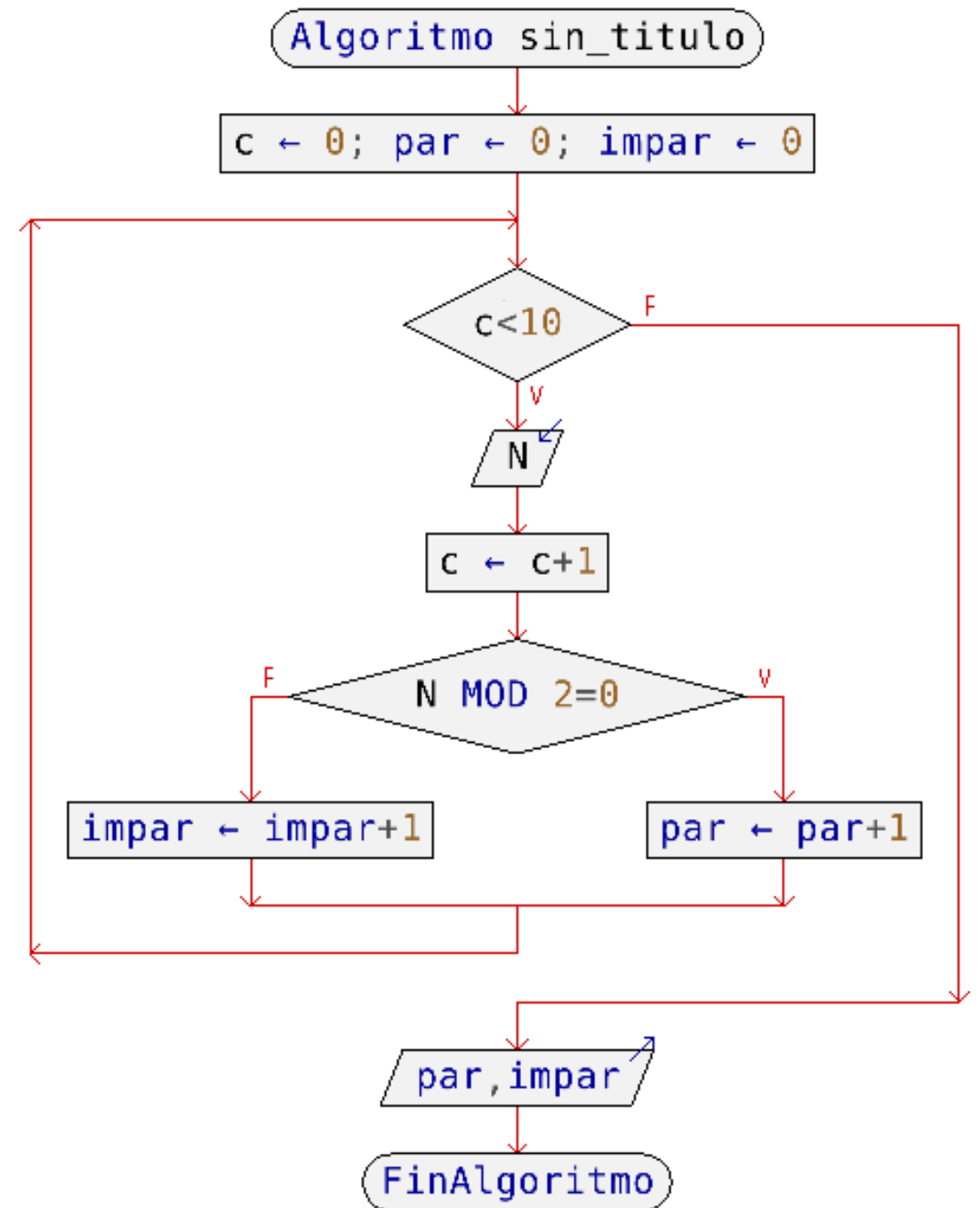
---

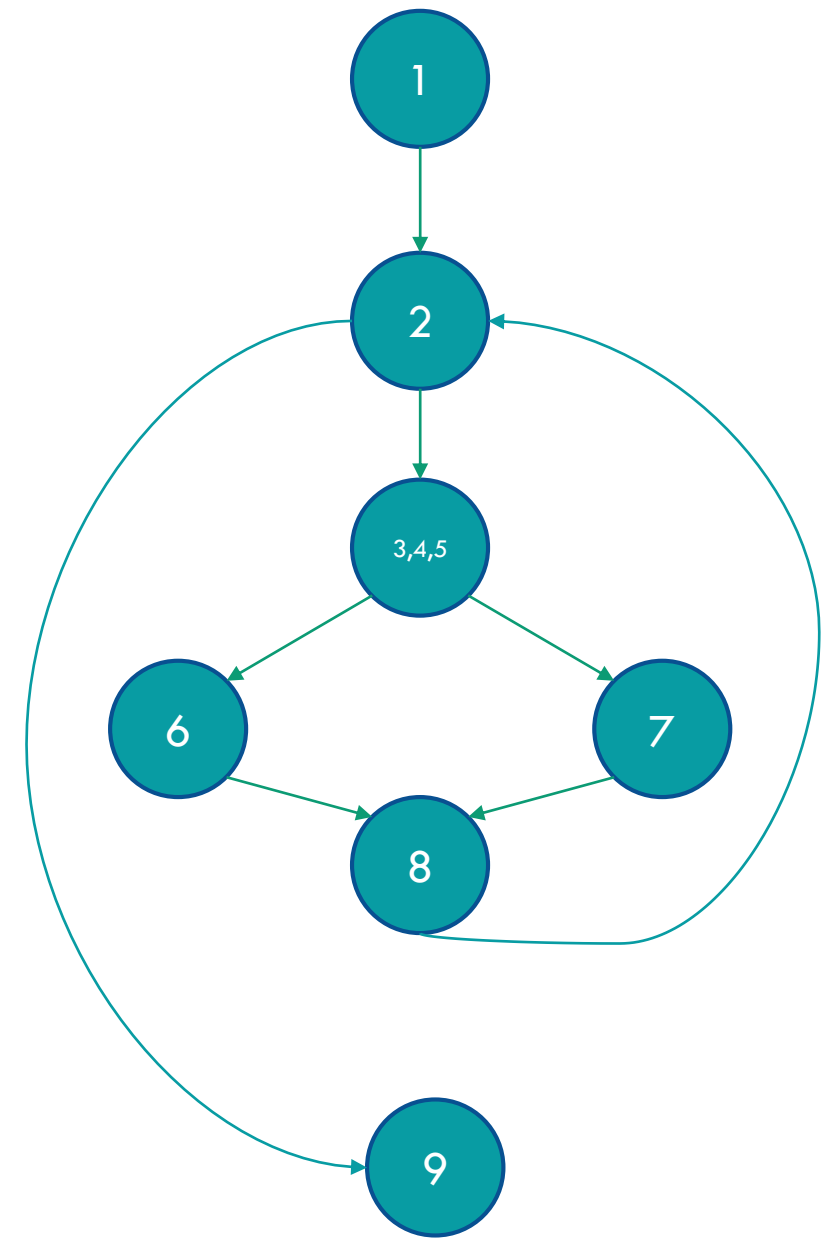
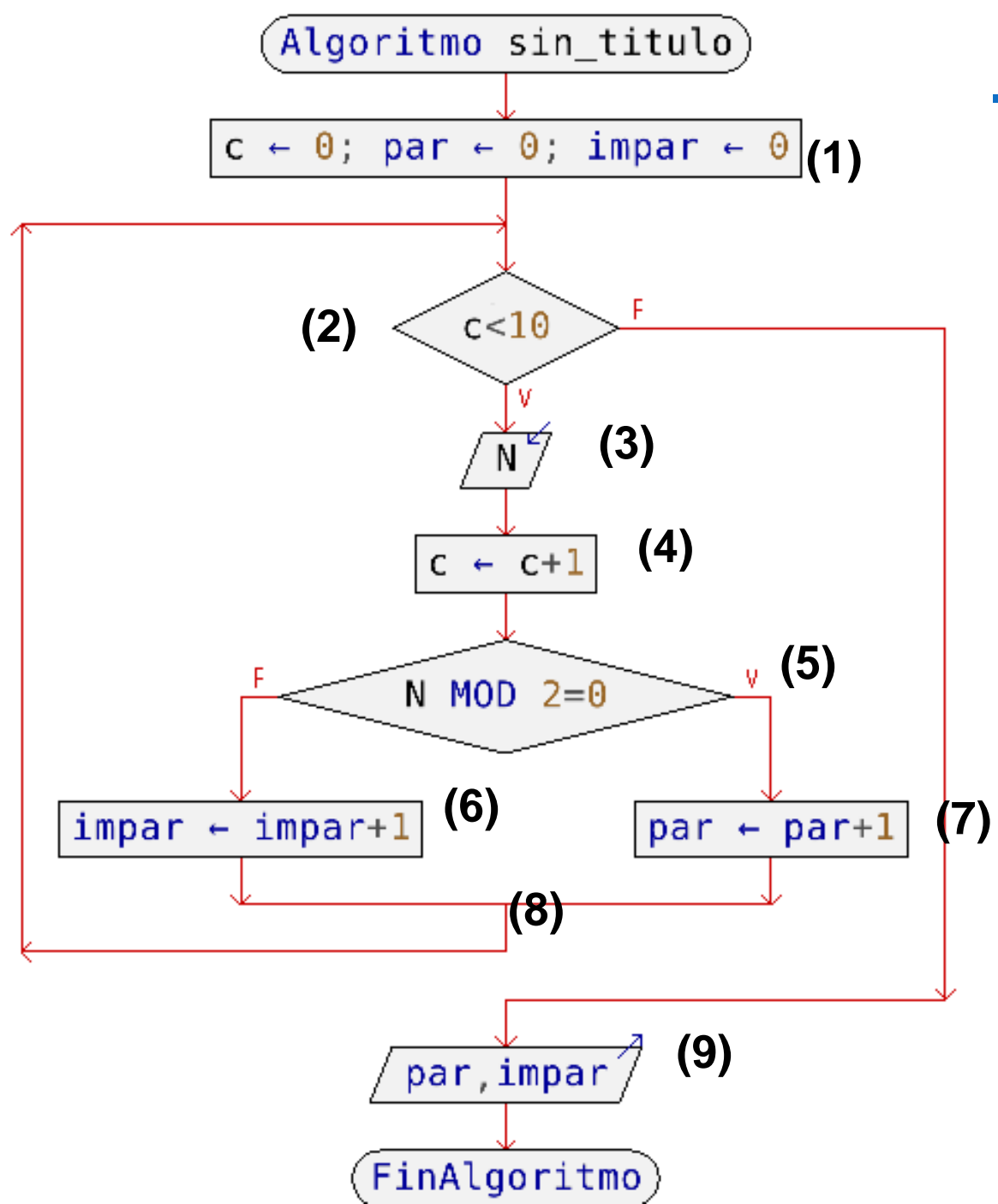
- Técnica de prueba de **caja blanca**
- Permite obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición .
- Garantizan la ejecución de cada sentencia del programa, al menos una vez.
- Notación de grafo de flujo.
  - **Nodos:** representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
  - **Aristas:** líneas que unen dos nodos.
  - **Regiones:** áreas delimitadas por aristas y nodos (área encerrada entre aristas y nodos). Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
  - **Nodos predicado:** nodo del que parten dos o más caminos.

Estructura	Grafo de flujo	Estructura	Grafo de flujo
<b>Secuencial</b> instrucción1 instrucción2 ... instrucciónN	 <pre> graph TD     1((1)) --&gt; 2((2)) </pre>	<b>Hacer mientras</b> mientras <condición1> <inst2> fin mientras3	 <pre> graph TD     1((1)) --&gt; 2((2))     2 --&gt; 3((3))     3 --&gt; 1 </pre>
<b>Condicional</b> si <condición1> entonces <instruc2> si no <instruc3> fin si4	 <pre> graph TD     1((1)) --&gt; 2((2))     1 --&gt; 3((3))     2 --&gt; 4((4))     3 --&gt; 4 </pre>	<b>Repetir hasta</b> Repetir <inst1> Hasta que <condición2>	 <pre> graph TD     1((1)) --&gt; 2((2))     2 --&gt; 3((3))     3 --&gt; 1 </pre>
<b>Condición múltiple</b> según sea <variable0> hacer caso opción1: <inst1> caso opción2: <inst2> otro caso3: <inst3> fin según4	 <pre> graph TD     0((0)) --&gt; 1((1))     0 --&gt; 2((2))     0 --&gt; 3((3))     1 --&gt; 4((4))     2 --&gt; 4     3 --&gt; 4 </pre>		

# Ejemplo 1

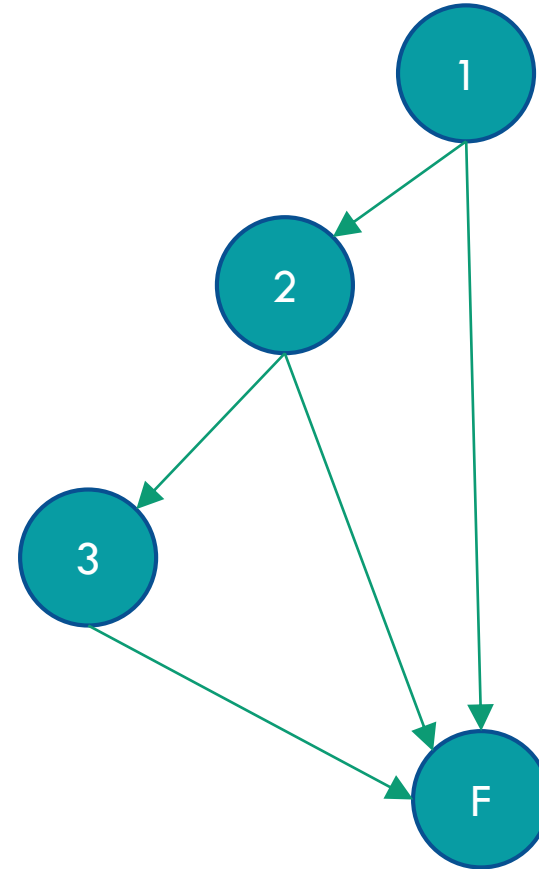
- Realiza el grafo de flujo a partir del ordinograma que se presenta a continuación.
- Se numeran cada uno de los símbolos. Cada uno corresponderá a un nodo.
- Se tendrán en cuenta los finales de los finales de las estructuras de control (if, while...)





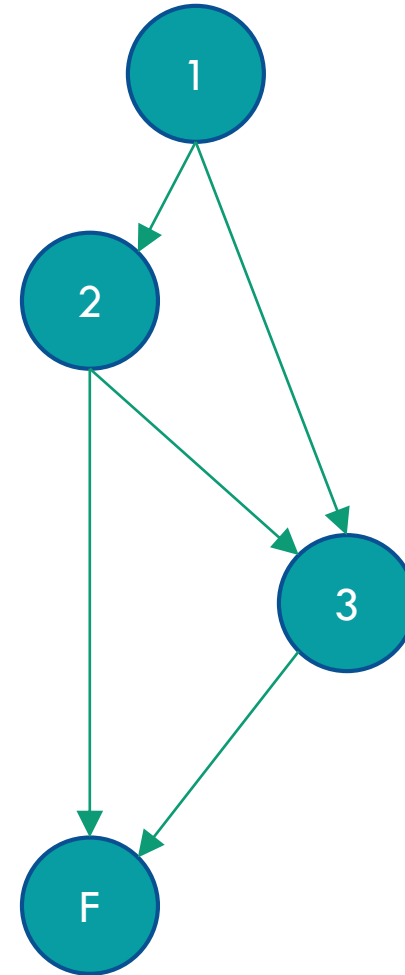
# Ejemplo de lógica compuesta: and

①  
si (**a** and **b**)  
③ sentencias 1  
F fin si



# Ejemplo de lógica compuesta: or

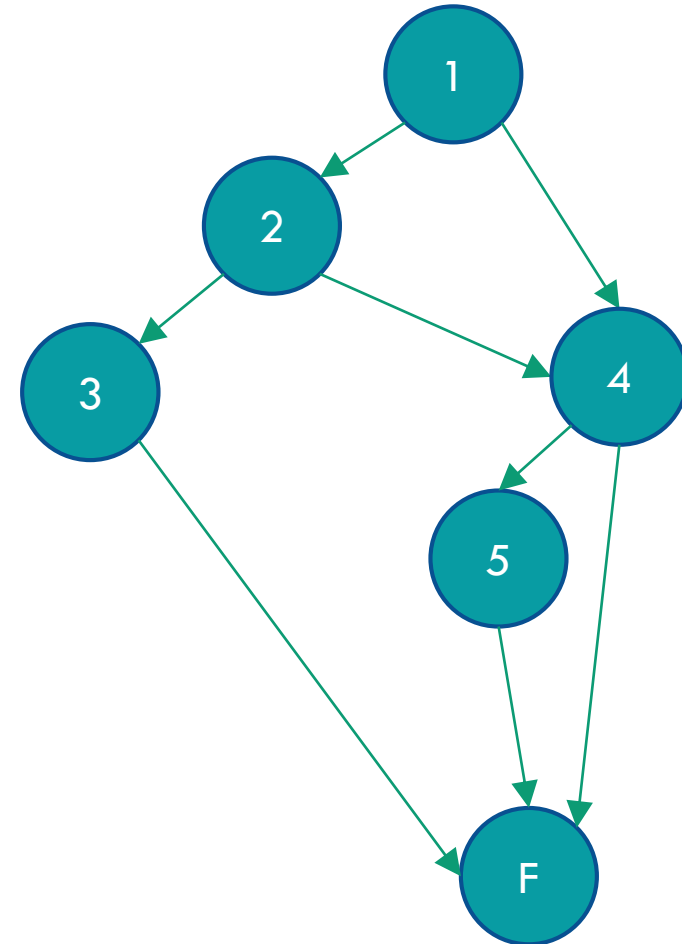
①                      ②  
si (a or b)  
③    sentencias1  
F   fin si





# Ejemplo de lógica compuesta

① si ② (a and b)  
③ sentencias1  
si no  
④ si c  
⑤ sentencias2  
fin si  
F fin si



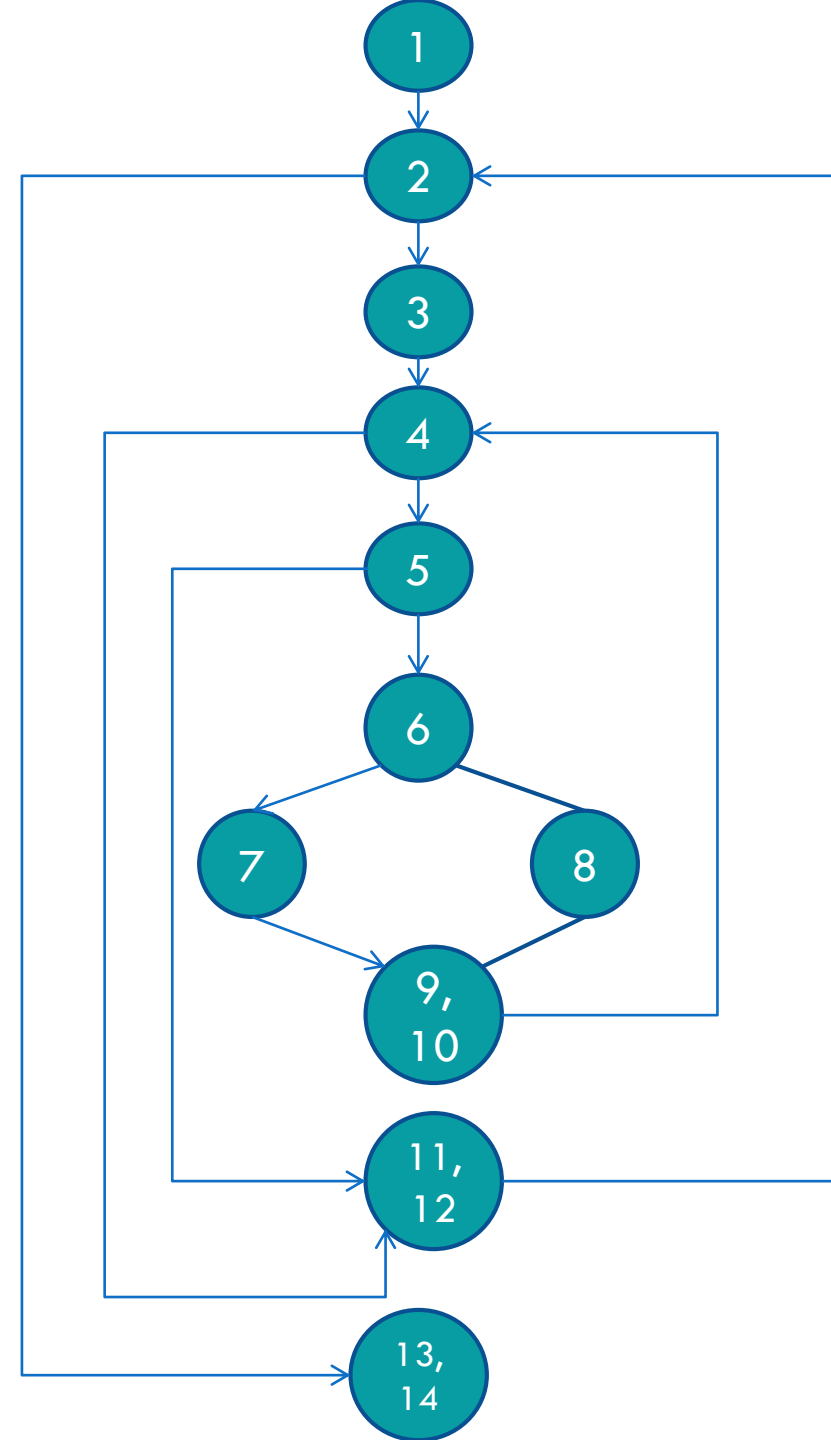
## Ejemplo 2

---

- Realiza el grafo de flujo a partir del pseudocódigo siguiente que se presenta a continuación.

```
Abrir archivo ventas;
Leer venta (producto, tipo_venta, total);
MIENTRAS (haya registros) HACER
    total_nacional=0;
    total_extranjero=0;
    MIENTRAS (haya registros) Y (mismo producto)
        SI (tipo_venta="nacional") ENTONCES
            total_nacional= total_nacional+ total;
        SI NO
            total_extranjero=total_extranjero+ total;
        FIN SI;
    Leer venta (producto, tipoVenta, total);
    FIN MIENTRAS;
    Escribir Producto, total_nacional, total_extranjero;
FIN MIENTRAS;
Cerrar archivo ventas;
```

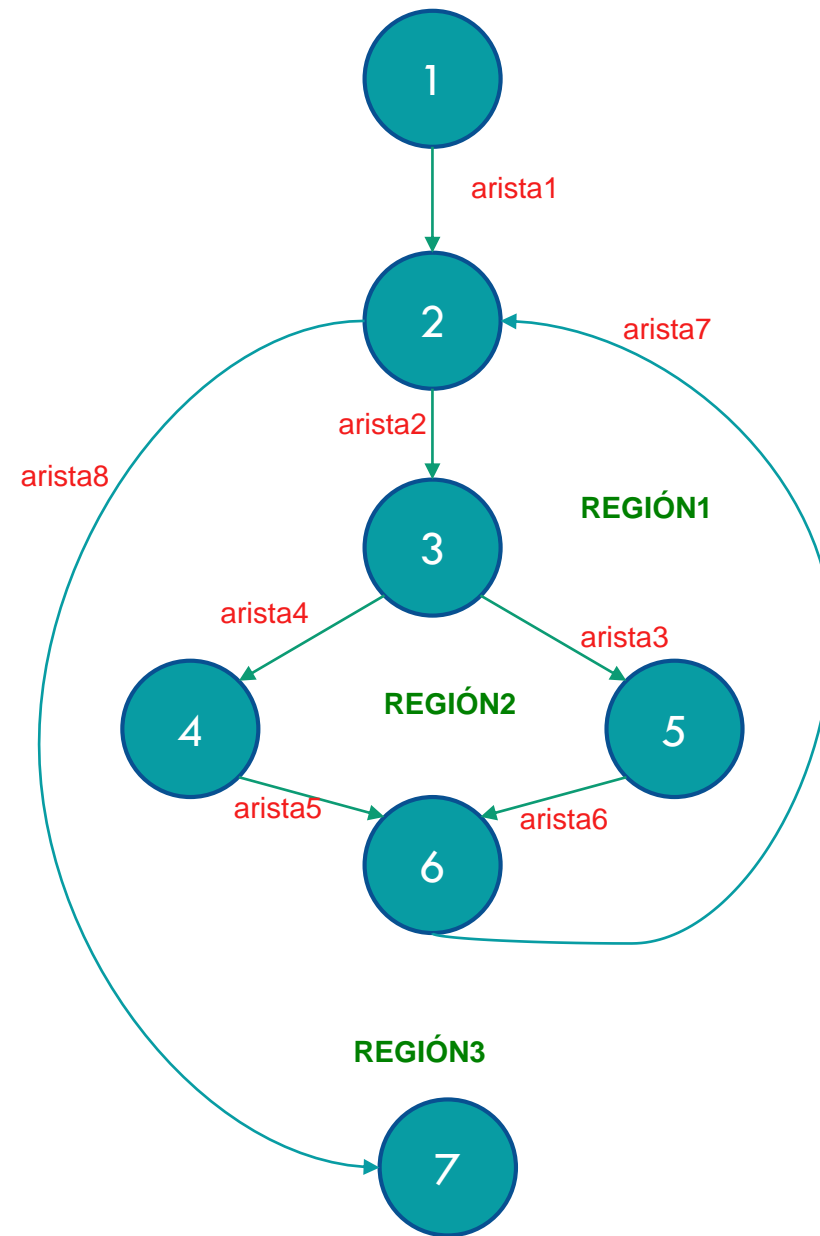
Abrir archivo ventas; (1)  
 Leer venta (producto, tipo\_venta, total); (2)  
 MIENTRAS (haya registros) HACER (3)  
     total\_nacional=0; (4)  
     total\_extranjero=0; (5)  
     MIENTRAS (haya registros) Y (mismo producto) (6)  
         SI (tipo\_venta="nacional") (7) ENTONCES  
             total\_nacional= total\_nacional+ total; (8)  
         SI NO  
             total\_extranjero=total\_extranjero+ total; (9)  
         FIN SI; (10)  
     Leer venta (producto, tipoVenta, total); (11)  
     FIN MIENTRAS; (12)  
     Escribir Producto, total\_nacional, total\_extranjero; (13)  
     FIN MIENTRAS; (14)  
     Cerrar archivo ventas; (15)



- Cálculo de la **complejidad ciclomática de McCabe**
  - Es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa.
  - Indicador del **número de caminos independientes** que existen en el grafo ➔ número de casos de prueba que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.
  - $V(G)$  se calcula de tres formas:
    - $V(G) = a - n + 2$ ; siendo **a** (nº de aristas), **n** (nº de nodos)
    - $V(G) = r$ ; siendo **r** (regiones cerradas del grafo)
    - $V(G) = c + 1$ ; siendo **c** (nº de nodos de condición)

# Ejemplo 1

- $V(G) = a - n + 2 = 8 - 7 + 2 = 3;$
- $V(G) = r = 3;$
- $V(G) = c + 1 = 2 + 1 = 3$
- Son 3 el número de caminos.
  - Camino1: 1 2 7
  - Camino2: 1 2 3 4 6 2 7
  - Camino3: 1 2 3 5 6 2 7
- Casos de prueba para los caminos



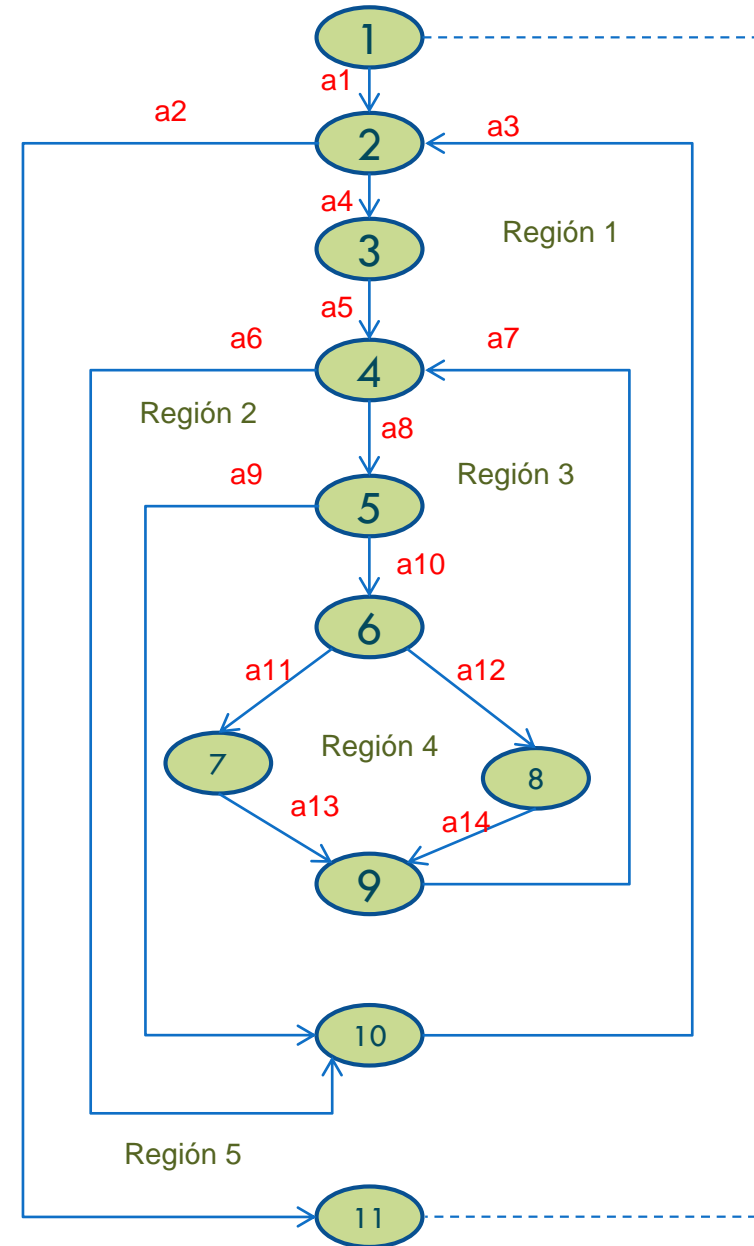
$$V(G) = 14 - 11 + 2 = 5.$$

$$V(G) = 5. \text{ Número de regiones} = 5$$

$$V(G) = 4 + 1. \text{ Nodos de condición: } 2, 4, 5, 6$$

Posible conjunto de caminos:

- 1-2-11
- 1-2-3-4-10-2
- 1-2-3-4-5-10-2
- 1-2-3-4-5-6-7-9-4-10-2-11
- 1-2-3-4-5-6-8-9-4-10-2-11
- Analizar el código para saber los datos de entrada necesarios para forzar la ejecución de cada uno de ellos



# Estructurales. Complejidad ciclomática de McCabe

- Se establecen los siguientes valores de la complejidad ciclomática.

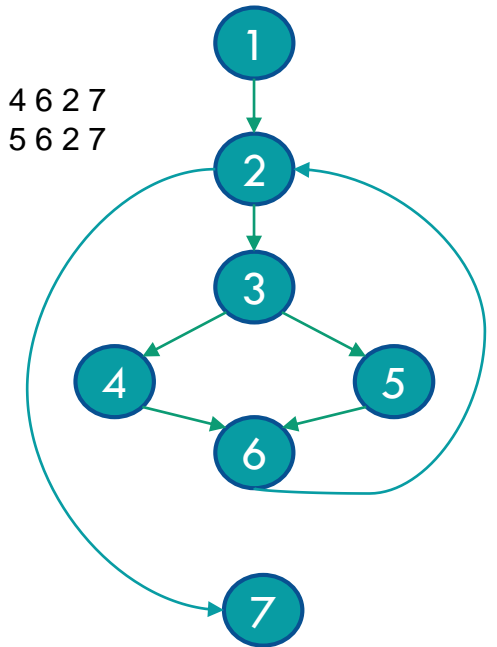
Complejidad ciclomática	Evaluación de riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado
Entre 21 y 50	Programas o métodos complejos, alto riesgo
Mayor que 50	Programas o métodos no testeables, muy alto riesgo

# Estructurales. Complejidad ciclomática de McCabe

- Obtención de los casos de prueba
  - Es el último paso a seguir.
  - Se trata de escoger los casos de prueba de forma que las condiciones de los nodos predicado estén adecuadamente establecidas.

Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que No se cumpla la condición $C < 10$	Visualizar el número de pares y el de impares
2	Escoger algún valor de C tal que Sí se cumpla la condición $C < 10$ Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$ $C = 1$ $N = 5$	Contar número impares
3	Escoger algún valor de C tal que Sí se cumpla la condición $C < 10$ Escoger algún valor de N tal que Sí se cumpla la condición $N \% 2 = 0$ $C = 2$ $N = 4$	Contar número pares

Camino1: 1 2 7  
Camino2: 1 2 3 4 6 2 7  
Camino3: 1 2 3 5 6 2 7





## Estructurales. Complejidad ciclomática de McCabe

---

- El camino 1 no puede ser probado por sí solo, debe ser probado como parte de las pruebas de los caminos 2, 3.

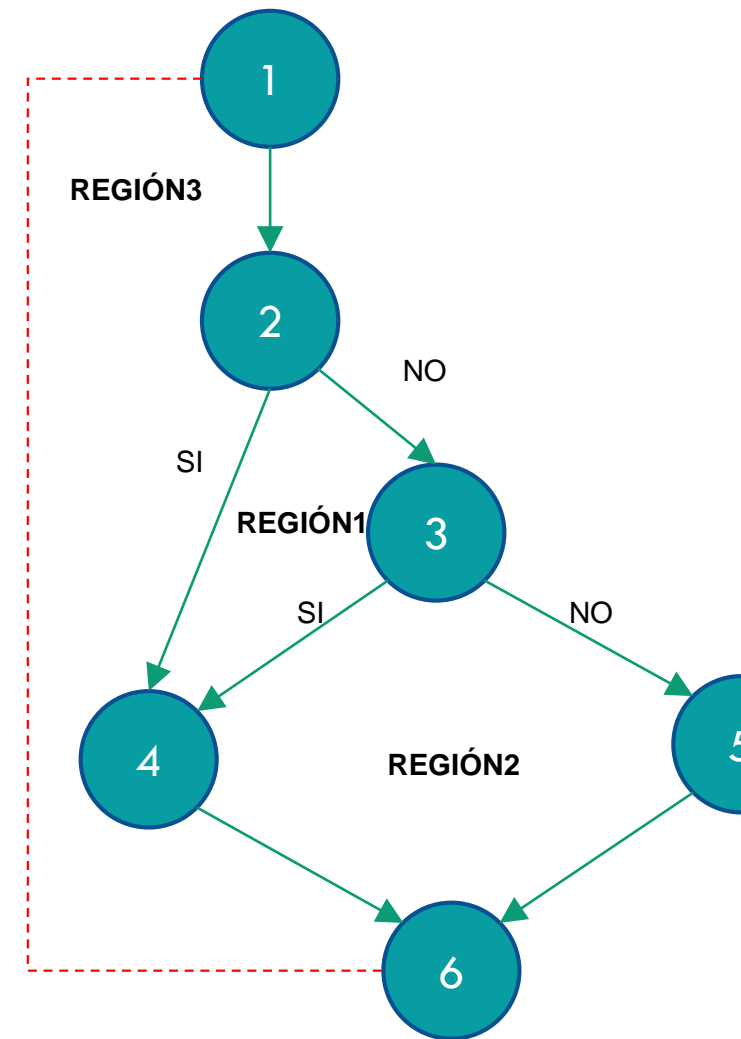
## Estructurales. Complejidad ciclomática de McCabe

- **Ejemplo:** realiza el grafo de flujo de la siguiente función Java, calcula la complejidad ciclomática de McCabe, los caminos independientes y casos de prueba

```
static void visualizaMedia(float x, float y){  
    float resultado = 0;  
    if (x<0 || y<0)  
        System.out.println("x e y deben ser positivo");  
    else{  
        resultado=(x+y)/2;  
        System.out.println("La media es: "+resultado);  
    }  
}
```

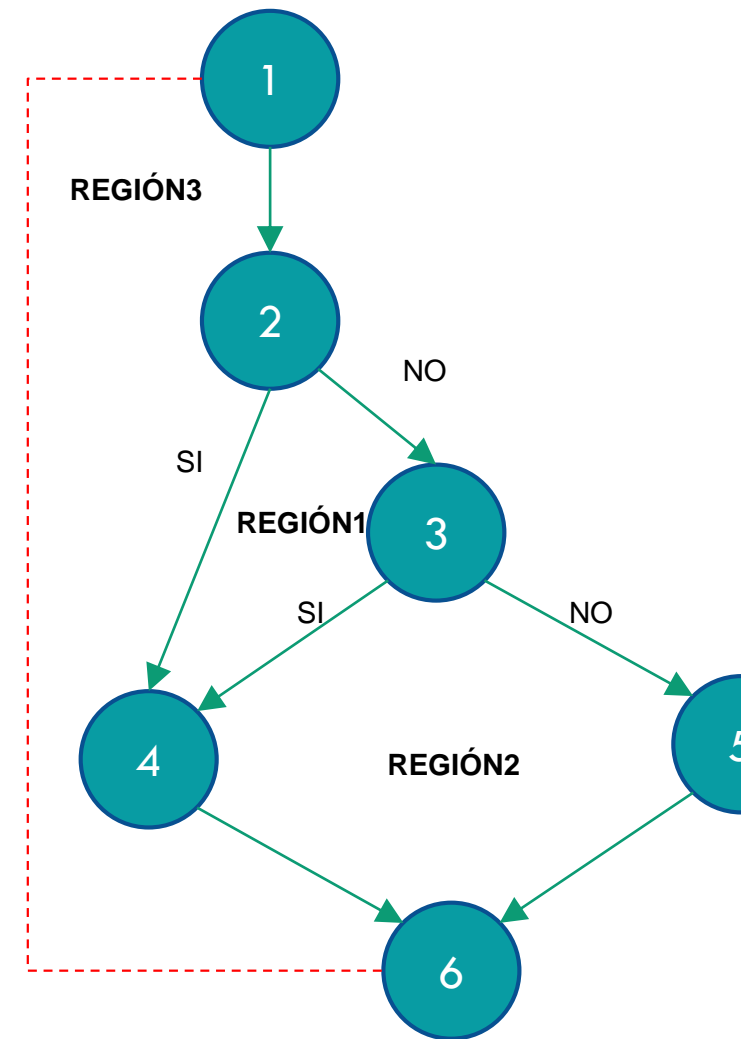
# Estructurales. Complejidad ciclomática de McCabe

```
static void visualizaMedia(float x, float y){  
    float resultado = 0; ①  
    if (x<0 || y<0) ③  
        System.out.println("x e y deben ser positivo"); ④  
    else{  
        resultado=(x+y)/2;  
        System.out.println("La media es: "+resultado); ⑤  
    }  
} ⑥
```



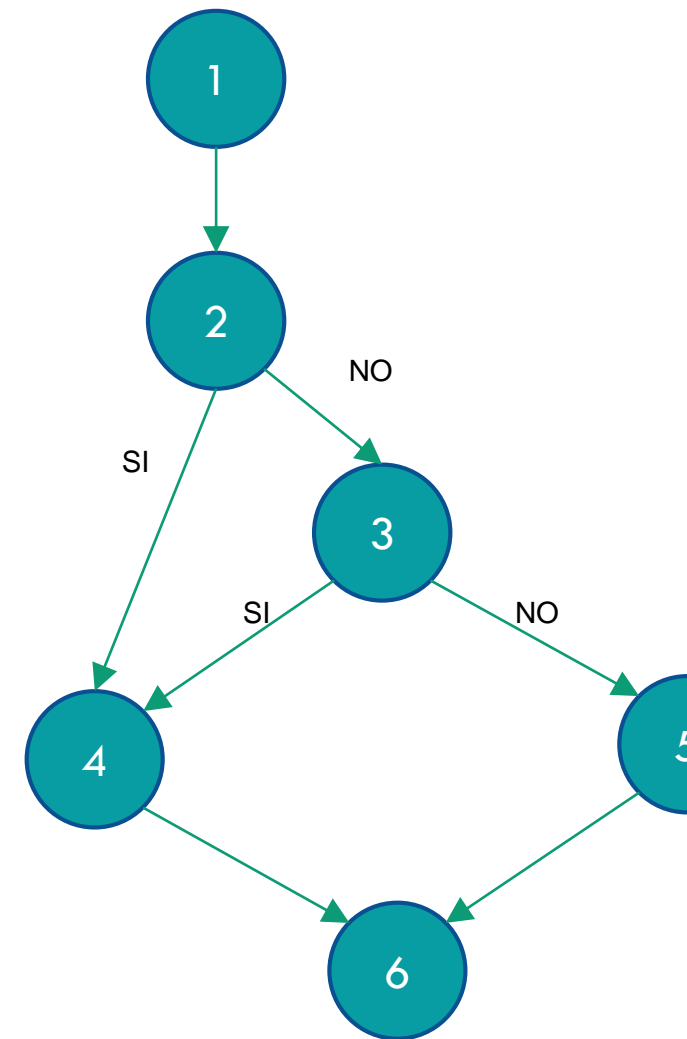
# Estructurales. Complejidad ciclomática de McCabe

- $V(G) = a - n + 2 = 7 - 6 + 2 = 3;$
- $V(G) = r = 3;$
- $V(G) = c + 1 = 2 + 1 = 3$
- Son 3 el número de caminos.
  - Camino1: 1 2 4 6
  - Camino2: 1 2 3 4 6
  - Camino3: 1 2 3 5 6
- Casos de prueba para los caminos

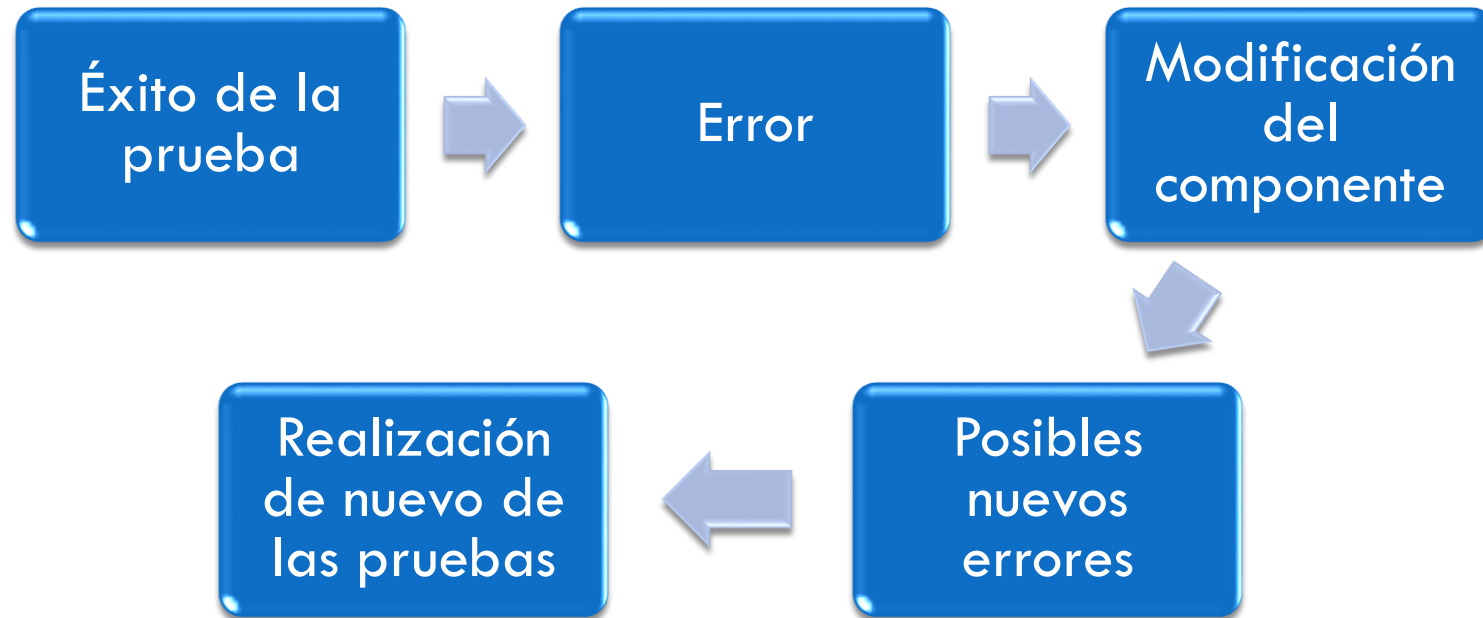


# Estructurales. Complejidad ciclométrica de McCabe

CAMINO	CASO DE PRUEBA	RESULTADO ESPERADO
Camino1 1 2 4 6	Escoger algún x tal que SÍ se cumpla la condición $x < 0$ (y, puede ser cualquier valor) $x = -4$ $y = 5$ <code>visualizarMedia(-4,5)</code>	x e y deben ser positivos
Camino2 1 2 3 4 6	Escoger algún x tal que NO se cumpla la condición $x < 0$ y escoger algún y que SÍ cumpla la condición $y < 0$ $x = 4$ $y = -5$ <code>visualizarMedia(4,-5)</code>	x e y deben ser positivos
Camino3 1 2 3 5 6	Escoger algún x tal que NO se cumpla la condición $x < 0$    $y < 0$ $x = 4$ $y = 5$ <code>visualizarMedia(4,5)</code>	La media es: 4.5



# Regresión



# Regresión

---

- Las pruebas de regresión → cada vez que se hace un **cambio** en el sistema (corregir un error o realizar una mejora).
- No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones **no produzcan efectos negativos** sobre el mismo u otros componentes.
- Implica la **repetición de las pruebas** que ya se hayan realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

# Regresión

---

- El conjunto de pruebas de regresión contiene tres clases diferentes de clases de prueba:
  - Una **muestra** representativa de pruebas que ejercite todas las funciones del software
  - **Pruebas adicionales** que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio
  - **Pruebas** que se centran en los componentes del software que han **cambiado**
- No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.