

## Sesión S02: Diseño de pruebas: **caja blanca**

Diseño de casos de prueba: structural testing

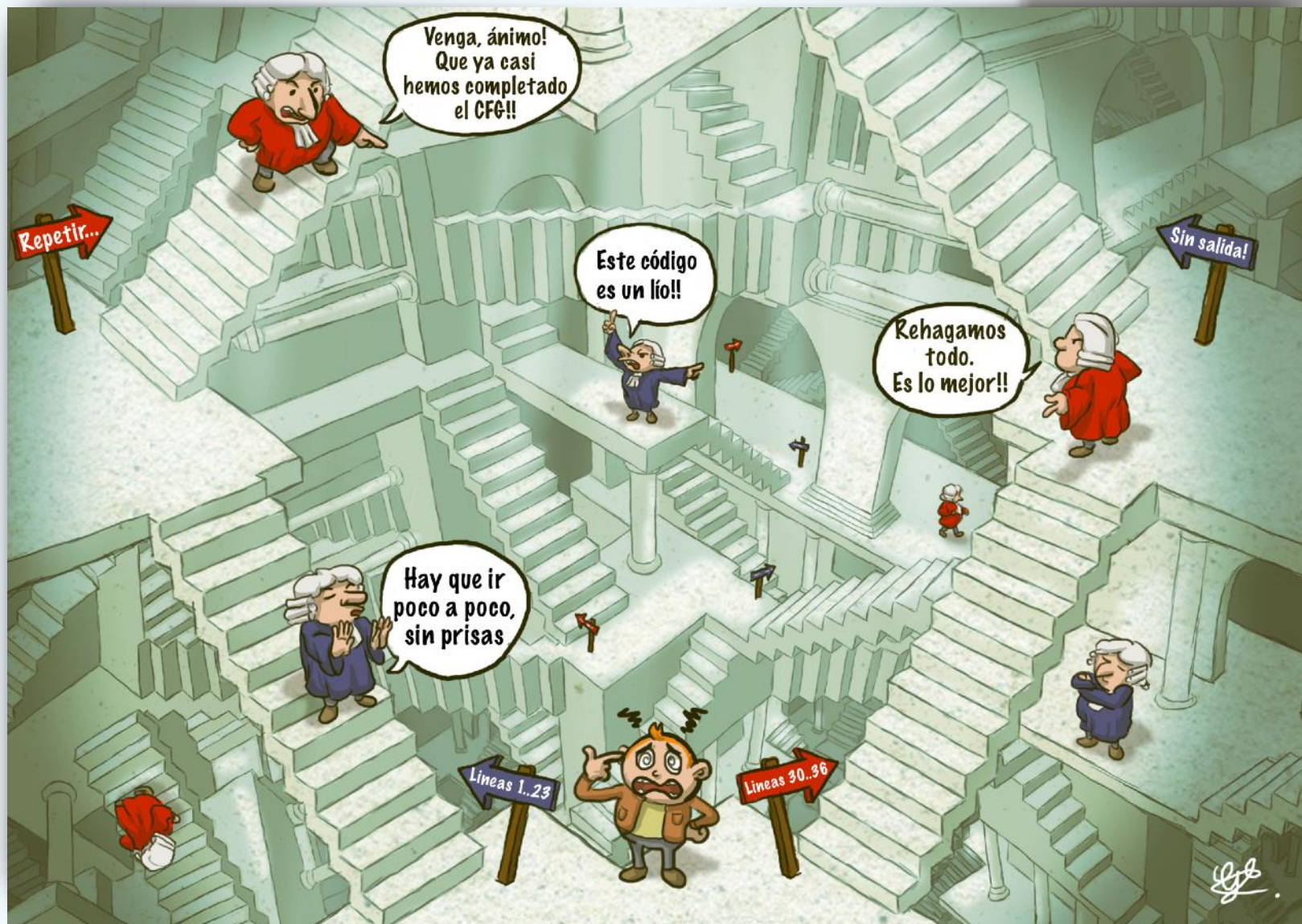
- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de **comportamientos programados**

- El conjunto obtenido debe detectar el máximo número posible de defectos en el código, con el mínimo número posible de "filas"

**Control flow testing:** Método del camino básico

- Paso 1: Análisis de código: Construcción del CFG
- Paso 2: Selección de caminos: Cálculo de CC y caminos independientes
- Paso 3. Obtención del conjunto de casos de prueba

Ejemplos y ejercicios





P

# DISEÑO DE CASOS DE PRUEBA

Selección de un subconjunto **mínimo** de entradas para evidenciar el **máximo** número de errores posibles

P

ACTIVIDADES DE PRUEBAS

PLANIFICACIÓN Y CONTROL

**diseño**

AUTOMATIZACIÓN

EVALUACIÓN

El resultado del proceso de diseño es una **Tabla de casos de prueba**. En el laboratorio hemos confeccionado una tabla de este tipo con casos de prueba que os hemos proporcionado, e incluso habéis añadido alguna fila

Tabla de casos de prueba

ID	d1	d2	...	Expected Output
C1	?	?	?	?
C2				
C3				
...				
<u>CN?</u>				

**Caso de prueba** =  
datos entrada  
concretos + resultado  
esperado

Cada FILA de la tabla es un  
CASO DE PRUEBA



- ❑ ¿COMO RELLENAMOS LAS FILAS DE LA TABLA?
- ❑ ¿CUÁNTAS FILAS ES NECESARIO AÑADIR?

Utilizando un **MÉTODO** de DISEÑO de casos de prueba!!!!



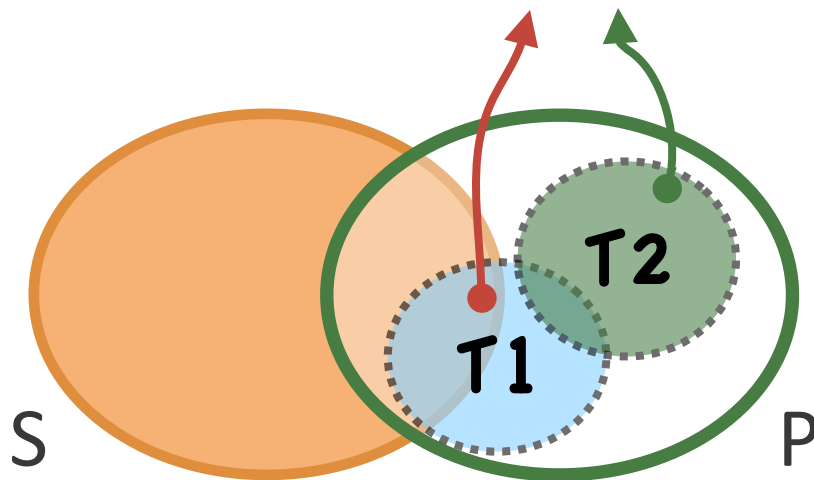
Es el proceso más importante, si queremos efectivamente alcanzar los objetivos planificados

# FORMAS DE IDENTIFICAR LOS CASOS DE PRUEBA

Caja BLANCA  
)))

## STRUCTURAL TESTING

Comportamientos probados



Podemos detectar comportamientos no especificados  
Nunca podremos detectar comportamientos no implementados

- ❑ Consiste en determinar los **valores de entrada** de los casos de prueba a partir de la IMPLEMENTACIÓN.
- ❑ El **resultado esperado** se obtiene SIEMPRE de la especificación
- ❑ Los comportamientos probados podrán estar o no especificados
- ❑ Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación

CUALQUIER MÉTODO BASADO EN EL CÓDIGO SIGUE ESTOS PRINCIPIOS!!!



Los métodos de diseño de casos de prueba basados en el CÓDIGO:

1. **Analizan** el código y obtienen una representación en forma de GRAFO
2. **Seleccionan** un conjunto de **camino**s en el grafo según algún criterio
3. **Obtienen** un conjunto de **casos de prueba** que ejercitan dichos caminos

Dependiendo del método utilizado, obtendremos conjuntos DIFERENTES de casos de prueba. Pero el conjunto obtenido será **EFFECTIVO** y **EFICIENTE!!!!**

Las técnicas o métodos estructurales son costosos de aplicar, por lo que suelen usarse sólo a nivel de **UNIDADES** de programa

## STRUCTURAL TESTING. OBSERVACIONES

DISEÑO

Los métodos estructurales **NO** pueden DETECTAR **TODOS** los defectos en el programa (**defecto** = **fault** = **bug**).

Aunque seleccionemos todas las posibles entradas, no podremos detectar todos los defectos si "faltan caminos" en el programa.

De forma intuitiva, diremos que falta un camino en el programa si no existe el código para manejar una determinada condición de entrada.

**Por ejemplo:** si la implementación no prevé que un divisor pueda tener un valor cero, entonces no se incluirá el código necesario para manejar esta situación



P

# EL ANÁLISIS DEL CÓDIGO SE PUEDE BASAR EN ...

P

## representar el FLUJO DE CONTROL

Hace referencia al flujo de control entre INSTRUCCIONES.

Podemos "visualizar" dicho flujo de control en un grafo dirigido.

Cada caso de prueba provoca la ejecución de un conjunto de instrucciones, representadas en el grafo como un "camino".

Un "camino", por lo tanto, es una determinada secuencia de acciones conducentes a obtener un resultado concreto (comportamiento)

A partir de una representación del flujo de control de las instrucciones, podemos:

- ▶ detectar de forma **DINÁMICA** defectos en el programa identificando diferentes caminos en el grafo y seleccionando aquellos que cumplan una determinada condición.

**Ejemplo** de método de diseño: McCabe's basis path method

## representar el FLUJO DE LOS DATOS

Hace referencia a la **propagación de valores** desde una variable (o constante) a otra variable. Las definiciones y usos de las variables determinan el flujo de datos en un programa.

A partir de una representación del flujo de valores de las variables, podemos:

- ▶ detectar de forma **ESTÁTICA** potenciales defectos del programa (anomalías de datos), identificando situaciones "anormales" que deben revisarse. Por ejemplo: variables definidas dos veces consecutivas
- ▶ detectar de forma **DINÁMICA** defectos en el programa identificando caminos en los que se manipulan las variables. Por ejemplo: ejecutar todos los caminos en los que una variable es definida y usada

Trabajaremos con representaciones de flujo de control !!!

# CONTROL FLOW TESTING

- Los dos tipos de sentencias básicas en un programa son.

- Sentencias de **asignación**  
Por defecto se ejecutan de forma secuencial
- Sentencias **condicionales**  
Alteran el flujo de control secuencial en un programa

- Las llamadas a "funciones" son un mecanismo para proporcionar abstracción en el diseño de un programa

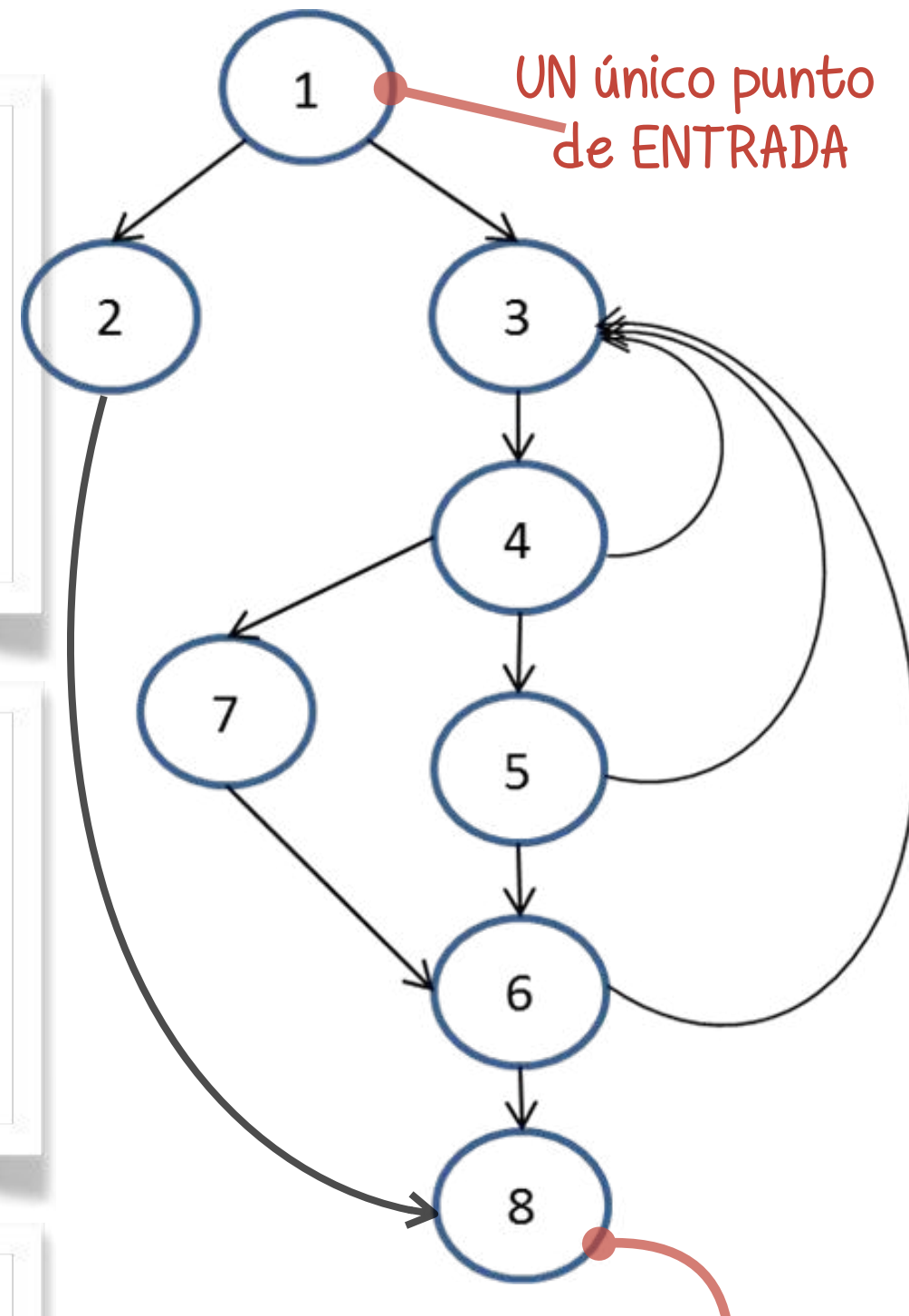
- Una llamada a una "función" cede el control a dicha función
- Dentro de la unidad a probar, la invocación de una "función" es una sentencia secuencial

Cada camino en el grafo se corresponde con un comportamiento!!!

La **EJECUCIÓN** de una secuencia de instrucciones desde el punto de entrada al de salida de una unidad de programa se denomina **CAMINO** (path)

En una unidad de programa puede haber un **número** potencialmente **grande** de caminos, incluso infinito.

Un **valor** específico de **entrada** provoca que se **ejecute** un camino específico en el programa



UNIDAD de programa = método Java

UN único punto de SALIDA

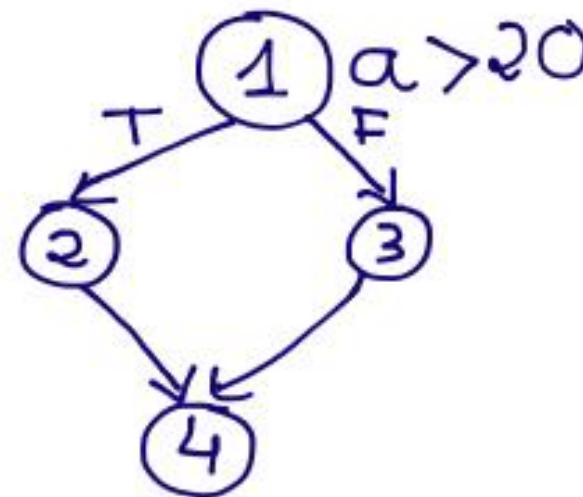
# GRAFO DE FLUJO DE CONTROL (CFG)

1 nodo representa cero o más sentencias secuenciales + cero o UNA!! condición

Un CFG es una **representación gráfica** de una **unidad** de programa. Usaremos un **GRAFO DIRIGIDO**, en donde:

- Cada **nodo** representa una o más sentencias secuenciales y/o una ÚNICA CONDICIÓN (así como los puntos de entrada y de salida de la unidad de programa)
  - \* Cada nodo estará etiquetado con un entero cuyo valor será único
  - \* Si un nodo contiene una condición anotaremos a su derecha dicha condición
- Las **aristas** representan el flujo de ejecución entre dos conjuntos de sentencias (representadas en los nodos)
  - \* Si uno nodo contiene una condición etiquetaremos las aristas que salen del nodo con "T" o "F" dependiendo de si el valor de la condición que representa es cierto o falso.

```
if (a > 20) {  
    k = "valor correcto"  
} else {  
    k = "repita entrada"  
}
```



# CONSTRUCCIÓN DE UN CFG (I)

Hay que tener claro cómo funcionan las sentencias de control del lenguaje!!!

Representa los grafos de flujo asociados a los siguientes códigos java:

```
if ((a > 1) && (a < 200)) {  
    ...  
}
```



Inténtalo!

```
1. if ((a != b) && (a != c) && (b != c)) {  
2.     ...  
3. } else {  
4.     if (a==b) {  
5.         if (a==c) {  
6.             ...  
7.         }  
8.     } else {  
9.         ...  
10.    }  
11. }
```



P

# CONSTRUCCIÓN DE UN CFG (II)



Puedes practicar con ejemplos sencillos para cada sentencia de control en Java!!!

P

- Fíjate que para poder crear correctamente el CFG necesitamos conocer BIEN el funcionamiento del subconjunto de sentencias de CONTROL del lenguaje de programación utilizado
- Por ejemplo, en Java se utilizan sentencias try..catch, para capturar y tratar las excepciones:

```
1. try {  
2.     s1; //puede lanzar Exception1;  
3.     s2; //no lanza ninguna excepción  
4.     ...  
5. } catch (Exception1 e) {  
6.     ...  
7. } finally {  
8.     ...  
9. }  
10. siguienteSentencia;
```



Inténtalo!

# CRITERIOS DE SELECCION DE CAMINOS

Recuerdas por qué tenemos que "seleccionar"??

- **Estructuralmente** un camino es una secuencia de instrucciones en una unidad de programa (desde el punto de entrada, hasta el punto de salida)
- **Semánticamente** un camino es una instancia de una ejecución de una unidad de programa (comportamiento programado)
- Es necesario **seleccionar** un conjunto de caminos con algún **criterio de selección**. Algunos ejemplos son:
  - Elegimos todos los caminos del grafo
  - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las SENTENCIAS al menos una vez
  - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las CONDICIONES al menos una vez
- No generaremos entradas para los tests en las que se ejecute el mismo camino varias veces. Aunque, si cada ejecución del camino actualiza el estado del sistema, entonces múltiples ejecuciones del mismo camino pueden no ser idénticas

Cada método de diseño usa un criterio de selección diferente!!!!  
Ese criterio depende de un OBJETIVO. Pero cualquier método de diseño proporciona un conjunto de casos de prueba efectivos y eficientes para ese objetivo!!!



# MCCABE'S BASIS PATH METHOD

(Método del camino básico)

- Es un método de DISEÑO de pruebas de caja blanca que permite ejercitar (ejecutar) cada camino independiente en el programa
  - Fue propuesto inicialmente por Tom McCabe en 1976. Este método permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esta medida como guía para la definición de un conjunto básico de caminos de ejecución
  - El método también se conoce como "Método del camino básico"

*El objetivo del método es éste!!!* 

- Si ejecutamos TODOS los caminos independientes, estaremos ejecutando TODAS las sentencias del programa, al menos una vez
  - Además estaremos garantizando que TODAS las condiciones se ejecutan en sus vertientes verdadero/falso

## ○ ¿Qué es un camino independiente?

- Es un camino en un grafo de flujo (CFG) que difiere de otros caminos en al menos un nuevo conjunto de sentencias y/o una nueva condición (Pressman, 2001)

*El número de caminos independientes determinará el número de filas de la tabla. Cada fila detectará defectos en un determinado subconjunto de sentencias del programa (ejercitando un determinado comportamiento)*



P

# DESCRIPCIÓN DEL MÉTODO

S

Recuerda que debes ser sistemático a la hora de aplicar los pasos y tener claro para qué estamos haciendo cada uno de ellos!!!

P

1. Construir el grafo de flujo del programa (CFG) a partir del código a probar
2. Calcular la complejidad ciclomática (CC) del grafo de flujo
3. Obtener los caminos independientes del grafo
4. Determinar los datos de prueba de entrada de la unidad a probar de forma que se ejerciten todos los caminos independientes
5. Determinar el resultado esperado para cada camino, en función de la ESPECIFICACIÓN de la unidad a probar

## Tabla resultante del diseño de las pruebas:

Camino	Entrada 1	Entrada 2	...	Entrada n	Resultado Esperado
C1	d11	d12	...	d1n	r1
...					
C2	d21	d22	...	d2n	r2

Recuerda que los valores de entrada y salida deben ser CONCRETOS!!!

Una columna para CADA dato de entrada

Una columna para CADA dato de salida

# COMPLEJIDAD CICLOMÁTICA

Determina el número de FILAS de la tabla!!

- Es una MÉTRICA que proporciona una medida de la **complejidad lógica** de un componente software

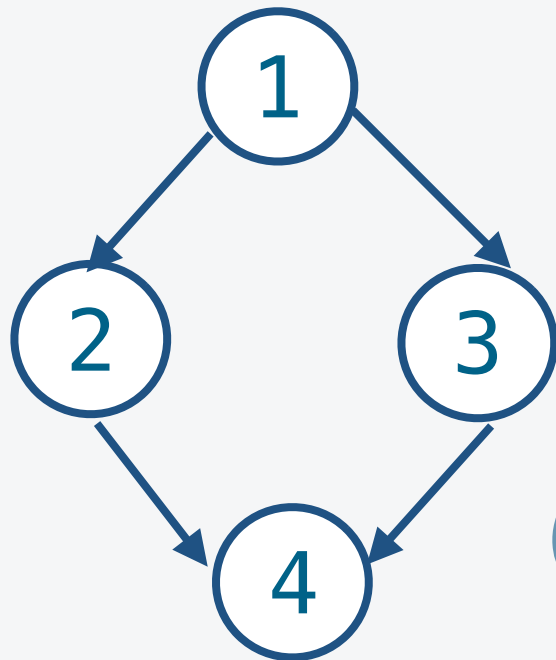
- Se calcula a partir del grafo de flujo:

$$CC = \text{número de arcos} - \text{número de nodos} + 2$$

- El valor de CC indica el MÁXIMO número de **caminos independientes** en el grafo

Un camino independiente aporta un nodo o una arista nuevas al conjunto de caminos

- Ejemplo:



$$CC = 4 - 4 + 2 = 2$$

Con 2 casos de prueba podemos recorrer todas las líneas y todas las condiciones en sus vertientes verdadera y falsa

A mayor CC, mayor complejidad lógica, por lo tanto, mayor esfuerzo de mantenimiento, y también mayor esfuerzo de pruebas!!!

El valor máximo de CC comúnmente aceptado como "tolerable" es 10

Se puede reducir la complejidad lógica refactorizando el código para incrementar el nivel de abstracción (modularizar)

# OTRAS FORMAS DE CALCULAR CC

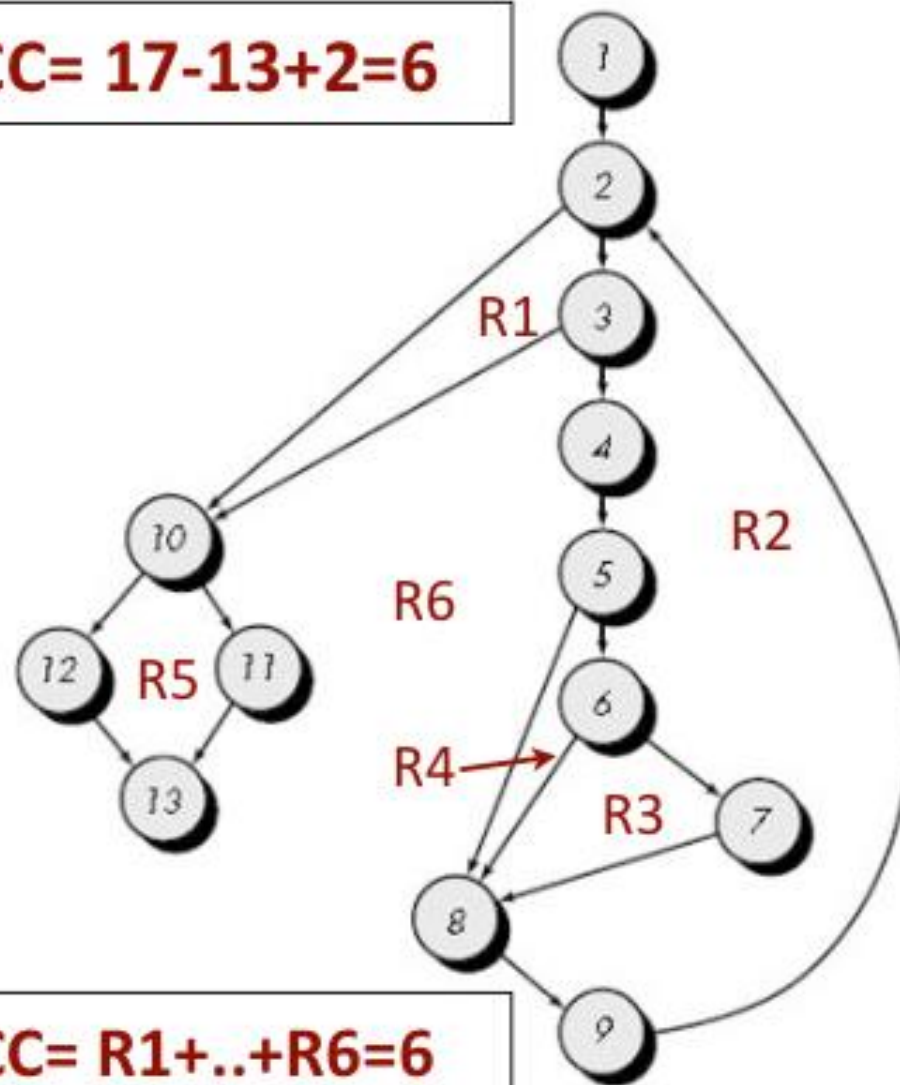
**CC** = número de arcos - número de nodos + 2

**CC** = número de regiones

**CC** = número de condiciones + 1

¡CUIDADO!: Las dos últimas formas de cálculo son aplicables SOLO si el código es totalmente estructurado (no saltos incondicionales)

$$CC = 17 - 13 + 2 = 6$$



$$CC = R1 + \dots + R6 = 6$$

```
...  
i=1;  
total.input=total.valid=0;  
sum=0;  
while ((value[i] <> -999) && (total.input<100)) {  
    total.input+=1;  
    if ((value[i]>= minimum) && (value[i]<= maximum)) {  
        total.valid+=1;  
        sum= sum + value[i];  
    }  
    i+=1;  
}  
if (total.valid >0) {  
    average= sum/total.valid;  
} else average = -999;  
return average;
```

$$CC = 5 + 1 = 6$$



P

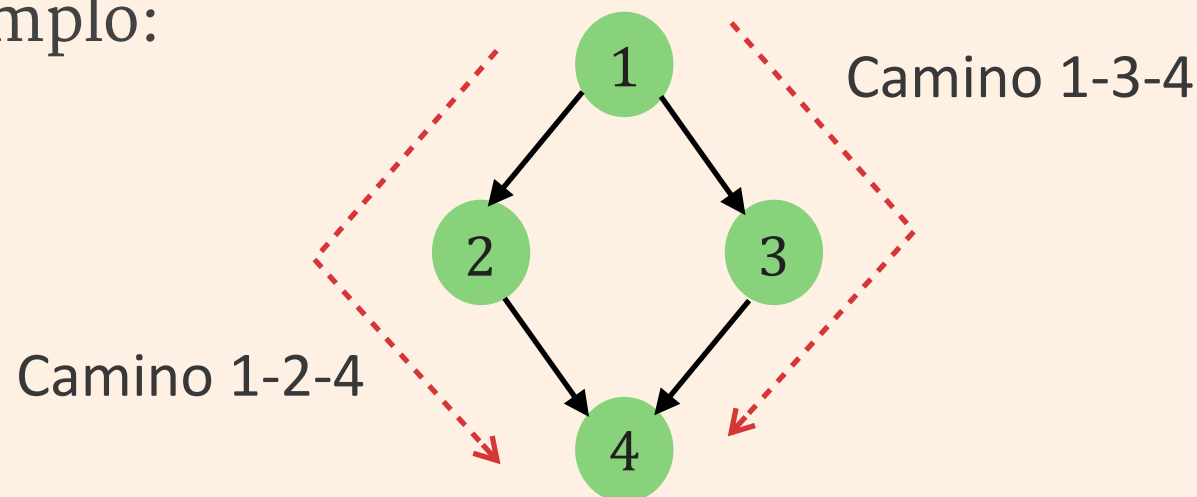
# CAMINOS INDEPENDIENTES

Cada camino independiente recorre un nodo o una arista (cómo mínimo) que no se había recorrido antes!!!

P

- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
  - Cada camino independiente contiene un nodo, o bien una arista, que no aparece en el resto de caminos independientes
  - Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

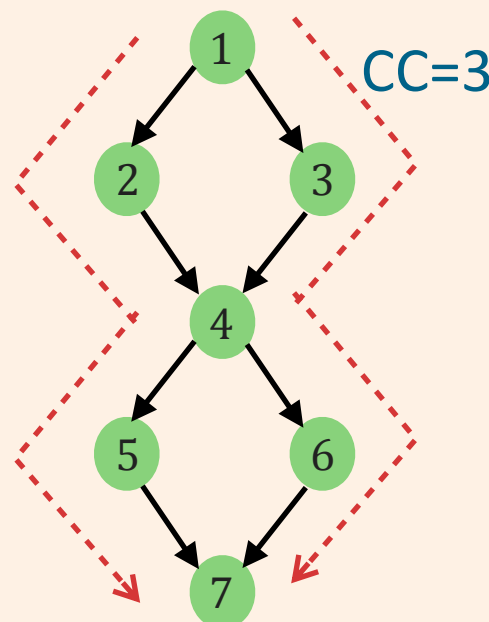
Ejemplo:



Es posible que con un número inferior a CC recorramos todos los nodos y todas las aristas

Ejemplo:

```
if (a >= 20) {  
    result = 0;  
} else {  
    result = 10;  
}  
if (b >= 20) {  
    result = 0;  
} else {  
    result = 10;  
}
```



opción 1:

C1 = 1-3-4-6-7  
C2 = 1-3-4-5-7  
C3 = 1-2-4-5-7

opción 2:

C1 = 1-3-4-6-7  
C2 = 1-2-4-5-7

Ambas opciones son válidas

P

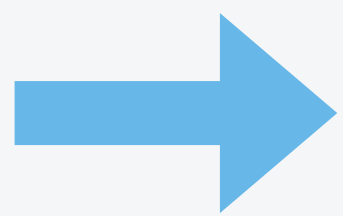
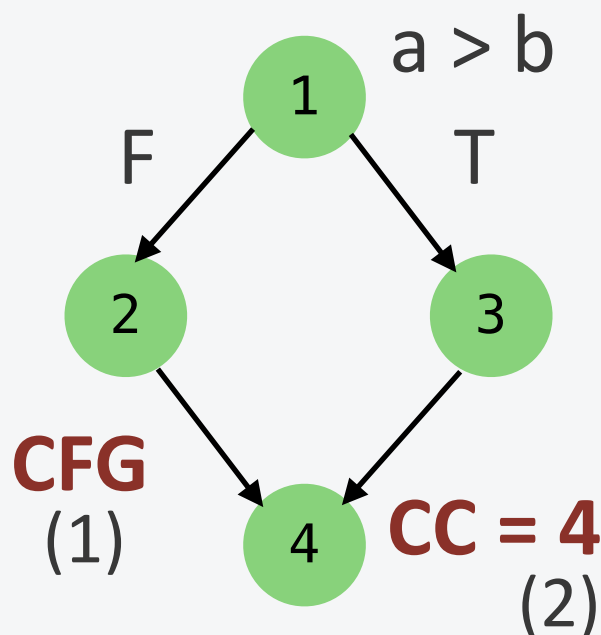
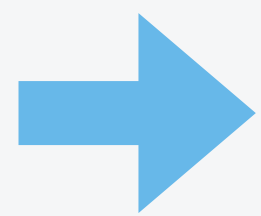
# EJEMPLO DE APLICACIÓN DEL MÉTODO

Es muy importante ser sistemático pero sabiendo lo que haces en cada paso!!!

P

Método que compara dos enteros a y b, y devuelve 20 en caso de que el valor a sea mayor que b, y cero en caso contrario:

```
if (a > b) {  
    result = 20;  
} else {  
    result = 0;  
}
```

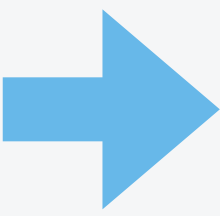


(3)  
**Caminos independientes**

C1 = 1-3-4  
C2 = 1-2-4

**CC = 4 - 4 + 2 = 2**  
(2)

Tabla resultante del diseño de casos de prueba



Camino	Datos Entrada		Resultado Esperado
C1	a = 20	b = 10	result = 20
C2	a = 10	b = 20	result = 0

(4) **Valores de entrada**

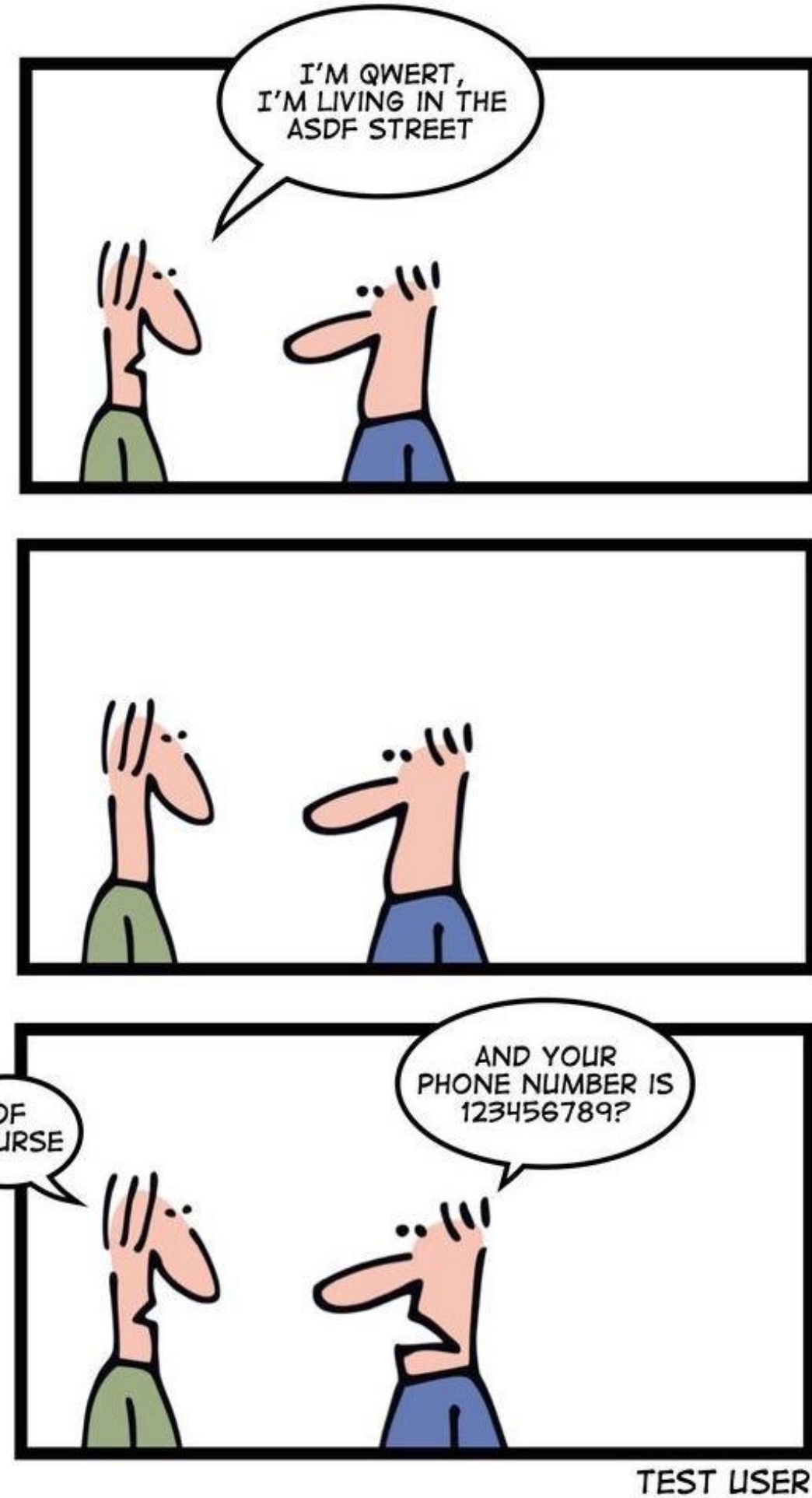
**Resultado esperado**

(5)

SIEMPRE son valores CONCRETOS!!!

# CONCRECIÓN DE LOS CASOS DE PRUEBA

- Recuerda que los datos de prueba y la salida esperada deben ser SIEMPRE valores CONCRETOS:
  - Los casos de prueba tienen que poder REPETIRSE, y por lo tanto, dos ejecuciones del mismo test, tienen que tener valores de entrada IDÉNTICAS
  - Por ejemplo:
    - \* D1= "cualquier dni válido" es INCORRECTO
    - \* D1= "12345678" es CORRECTO
- ¿Por qué repetir los tests?
  - Piensa en razones por las que va a ser necesario repetir los tests
  - Piensa en las consecuencias de no repetir los tests







# OBSERVACIONES SOBRE LOS DATOS DE PRUEBA



- Normalmente los datos de ENTRADA y/o SALIDA podremos identificarlos en los parámetros del elemento a probar, (y/o valores de retorno). Pero esto NO siempre es así:
  - Por ejemplo, supongamos que queremos probar el siguiente método:
    - \* **nuevo\_cliente(cliente)** añade un nuevo cliente a lista de clientes de nuestra tienda virtual, después de comprobar que el cliente que se pasa como entrada no es cliente de la tienda. Si el cliente ya existe el método no hará nada. Supongamos que la signatura del método es:
    - \* **public void nuevo\_cliente(Client cli)**, ¿cuáles son las entradas y salidas?
    - \* En este caso, el estado de la lista de clientes "antes" de llamar al método lo tenemos que considerar como una entrada, y el estado de la lista de clientes "después" de llamar al método lo consideraremos una salida
- Si utilizamos un lenguaje orientado a objetos, como Java, consideraremos como datos de entrada los valores concretos de "cada atributo" del objeto
  - Por ejemplo, supongamos que la clase Cliente está formada por los campos dni, nombre, dirección, y teléfonos. Un ejemplo de dato de entrada para un Cliente podría ser éste:
    - \* dni=12345678, nombre= "pepe", dirección="calle del mar", teléfonos=(12345,99999)

# EJEMPLO: BÚSQUEDA BINARIA

//Asumimos que la lista de elementos está ordenada de forma ascendente

```
class BinSearch
public static void search (int key, int [ ] elemArray, Result r) {
    int bottom = 0;    int top = elemArray.length -1;
    int mid;    r.found= false; r.index= -1;
    while (bottom <= top) {
        mid = (top+bottom)/2;
        if (elemArray [mid] == key) {
            r.index = mid;
            r.found = true;
            return;
        } else {
            if (elemArray [mid] < key)
                bottom = mid + 1;
            else top = mid -1;
        }
    } //while loop
} //search
} //class
```

Especificación del método search():  
Dado un vector de enteros ordenados ascendentemente, y dado un entero (key) como entrada, el método search() busca la posición de key en el vector y devuelve el valor found=true si lo encuentra, así como su posición en el vector (dada por index). Si el valor de key no está en el vector, entonces devuelve el valor found=false

# VAMOS A IDENTIFICAR LOS NODOS DEL GRAFO

//Asumimos que la lista de elementos está ordenada de forma ascendente

**class BinSearch**

**public static void search** (int key, int [ ] elemArray, Result r)

```
{  int bottom = 0;      int top = elemArray.length -1;
  int mid;              r.found= false; r.index= -1;
```

```
  while (bottom <= top) {
```

```
    mid = (top+bottom)/2;
```

```
    if (elemArray [mid] == key) {
```

```
      r.index = mid;
```

```
      r.found = true;
```

```
      return;
```

```
    } else {
```

```
      if (elemArray [mid] < key)
```

```
        bottom = mid + 1;
```

```
      else top = mid -1;
```

```
    }
```

```
  } //while loop
```

```
} //search
```

```
} //class
```

1

2

3

8

4

5

6

7

9

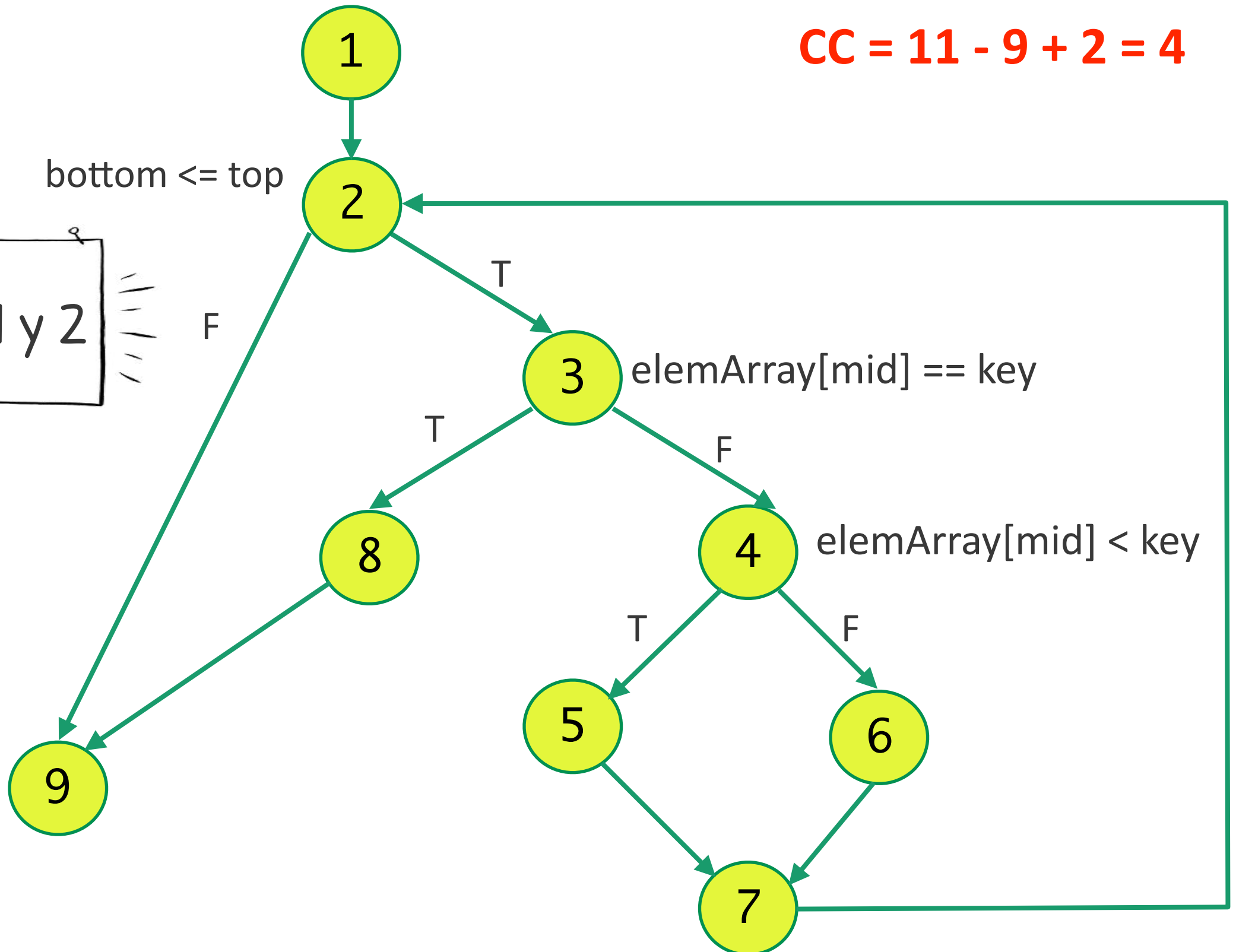


# GRAFO ASOCIADO Y VALOR DE CC

$$CC = 11 - 9 + 2 = 4$$

bottom <= top

Pasos 1 y 2



P

# CAMINOS INDEPENDIENTES

S

○ Posible conjunto de caminos independientes

P

- ❑ C1: 1, 2, 3, 4, 6, 7, 2, 9
- ❑ C2: 1, 2, 3, 4, 5, 7, 2, 9
- ❑ C3: 1, 2, 3, 8, 9

Paso 3

En este ejemplo, con tres caminos podemos recorrer todos los nodos y todas las aristas

○ Ejercicio: Calcula la tabla resultante:

Camino	Datos Entrada		Resultado Esperado	
	key	elemArray	r.found	r.index
C1				
C2				
C3				

Pasos 4 y 5

Para indicar el valor del resultado esperado necesitamos conocer la ESPECIFICACIÓN del método



P

# EJERCICIOS PROPUESTOS (I)

S

P

● Calcula la CC para cada uno de estos códigos Java:

```
public void divide(int numberToDivide, int numberToDivideBy) throws BadNumberException{
    if(numberToDivideBy == 0){
        throw new BadNumberException("Cannot divide by 0");
    }
    return numberToDivide / numberToDivideBy;
}
```

Código 1

```
public void callDivide(){
    try {
        int result = divide(2,1);
        System.out.println(result);
    } catch (BadNumberException e) {
        //do something clever with the exception
        System.out.println(e.getMessage());
    }
    System.out.println("Division attempt done");
}
```

Código 2

```
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1){
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        }
        reader.close();
        System.out.println("--- File End ---");
    } catch (FileNotFoundException e) {
        //do something clever with the exception
    } catch (IOException e) {
        //do something clever with the exception
    }
}
```

Código 3




# EJERCICIOS PROPUESTOS (II)

Diseña los casos de prueba para el método validar\_PIN(), cuyo código es el siguiente:

```
1.  public class Cajero {
2.  ...
3.  public boolean validar_PIN (Pin pinNumber) {
4.      boolean pin_valido= false;
5.      String codigo_respuesta="GOOD";
6.      int contador_pin= 0;
7.
8.      while ((!pin_valido) && (contador_pin <= 2) &&
9.              !codigo_respuesta.equals("CANCEL")) {
10.         codigo_respuesta = obtener_pin(Pin pinNumber);
11.         if (!codigo_respuesta.equals("CANCEL")) {
12.             pin_valido = comprobar_pin(pinNumber);
13.             if (!pin_valido) {
14.                 System.out.println("PIN inválido, repita");
15.                 contador_pin=contador_pin+1;
16.             }
17.         }
18.     }
19.     return pin_valido;
20. }
21. ...
22. }
```

la especificación la tenéis a continuación



P

# EJERCICIOS PROPUESTOS (II) (CONTINUACIÓN)

Especificación del método `validar_PIN()`:

P

- ❑ El método `validar_PIN()` anterior valida un código numérico de cuatro cifras (objeto de la clase `Pin`). Dicho código se obtendrá después de introducirlo a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar). Si el usuario pulsa en algún momento la tecla de cancelar, entonces la validación se considerará "false". El usuario dispone de tres intentos para introducir un pin válido, en cuyo caso el método `validar_PIN()` devuelve cierto, así como el número de pin, y en caso contrario devuelve falso.

**Nota:** La introducción del código numérico se ha implementado en otra unidad (el método `obtener_pin()`, que se encargará de “leer” el código introducido por teclado creando una nueva instancia de un objeto `Pin`, y devuelve “GOOD” si no se pulsa la tecla para cancelar, o “CANCEL” si se ha pulsado la tecla para cancelar (carácter ‘\’).

Además usamos el método `comprobar_pin()`, que verifica que el código introducido tiene cuatro cifras y se corresponde con la contraseña almacenada en el sistema para dicho usuario, devolviendo cierto o falso, en función de ello.

Recuerda que el método del camino básico sólo lo aplicaremos a nivel de UNIDAD!!



Hemos definido una unidad como  
UNIDAD = MÉTODO JAVA

P  
P

# Y AHORA VAMOS AL LABORATORIO...

El proceso de diseño lo haremos "manualmente"

Hay que tener claros TODOS los pasos!!!

Diseñaremos casos de prueba utilizando el método del CAMINO BÁSICO

```
package ppss;

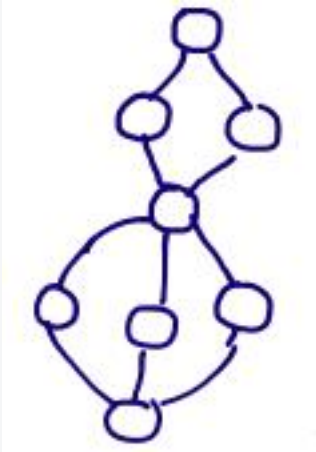
public class Matricula {
    public float calculaTasaMatricula(int edad,
        boolean familiaNumerosa,
        boolean repetidor) {
        float tasa = 500.00f;

        if ((edad <= 25) && (!familiaNumerosa) || (!repetidor)) {
            tasa = tasa + 1500.00f;
        } else {
            if ((familiaNumerosa) || (edad > 65)) {
                tasa = tasa / 2;
            }
            if ((edad >= 50) && (edad < 65)) {
                tasa = tasa - 100.00f;
            }
        }

        return tasa;
    }
}
```

unidad a probar

CFG



CC

CC = ...

tabla de casos de prueba

caminos independientes

- C1: 1-2-4-... -14
  - C2: 1-3-6-... -14
  - ...
  - CN: 1-2-7-... -14
- (N ≤ CC)

Camino	DATOS DE ENTRADA				RESULTADO ESPERADO		
C1	d11	d12	...	d1q	r11	...	r1k
...							
CN	dn1	dn2	...	dnq	r n1		r nk

ESTA TABLA La utilizaremos en la siguiente práctica!!!... 26





# REFERENCIAS BIBLIOGRÁFICAS



- A practitioner's guide to software test design. Lee Copeland. Artech House Publishers. 2004
  - Capítulo 10: Control Flow Testing
- Pragmatic software testing. Rex Black. Wiley. 2007
  - Capítulo 21: Control Flow Testing
- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
  - Capítulo 4: Control Flow Testing