Testing Estructural

Algoritmos y Estructuras de Datos I

¿Qué es hacer testing?

- Es el proceso de ejecutar un producto para ...
  - Verificar que satisface los requerimientos (en nuestro caso, la especificación)
  - Identificar diferencias entre el comportamiento **real** y el comportamiento **esperado** (IEEE Standard for Software Test Documentation, 1983).
- Objetivo: encontrar defectos en el software.
- Representa entre el 30 % al 50 % del costo de un software confiable.

Motivación

- Sea la siguiente especificación

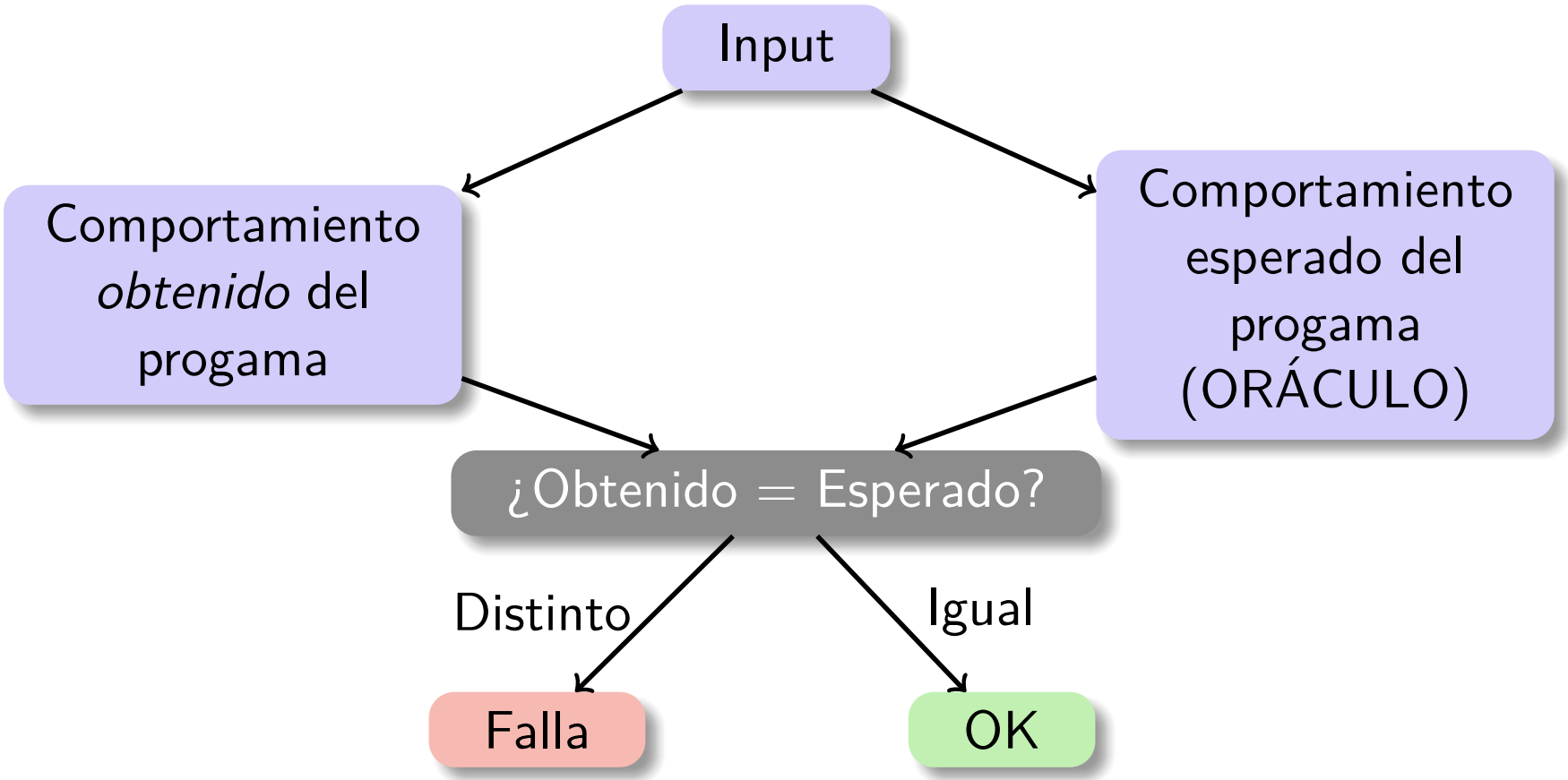
  proc *primo*(in $n : \mathbb{Z}$, out *result* : Bool) {
  Pre $\{n > 1\}$
  Post $\{result = true \leftrightarrow \text{esPrimo}(n)\}$
  }

- Si tenemos un programa, lo probamos, para el número 17, y nos devuelve: `true`. ¿está bien?

```
bool primo(int n){
  return true;
}
```

Este programa cumple con la especificación para todos los números que efectivamente son primos, y para los demás falla.

¿Cómo se hace testing?



Definiciones

Test Input, Test Case y Test Suite

- **Programa bajo test**: Es el programa que queremos saber si funciona bien o no.
- **Test Input** (o dato de prueba): Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.
- **Test Case:** Caso de Test (o caso de prueba). Es un programa que ejecuta el programa bajo test usando un dato de test, y chequea (automáticamente) si se cumple la condición de aceptación sobre la salida del programa bajo test.
- **Test Suite**: Es un conjunto de casos de Test (o de conjunto de casos de prueba).

Hagamos Testing

- ¿Cuál es el programa bajo test?
  - Es la implementación de una especificación.
- ¿Entre qué datos de prueba puedo elegir?
  - Aquellos que cumplen la **precondición** en la especificación
- ¿Qué condición de aceptación tengo que chequear?
  - La condición que me indica la **postcondición** en la especificación.
- ¿Qué pasa si el dato de prueba no satisface la precondición de la especificación?
  - Entonces no tenemos ninguna condición de aceptación

Especificando un semáforo

- Representamos con tres valores de tipo *Bool* el estado de la luz verde, amarilla y roja de un semáforo.
- Podemos especificar un predicado para representar cada estado válido del semáforo:

pred esRojo(v, a, r: Bool) { $v = \textbf{false} \land a = \textbf{false} \land r = \textbf{true}$ }

pred esAmarillo(v, a, r: Bool) { $v = \textbf{false} \land a = \textbf{true} \land r = \textbf{false}$ }

pred esVerde(v, a, r: Bool) { $v = \textbf{true} \land a = \textbf{false} \land r = \textbf{false}$ }

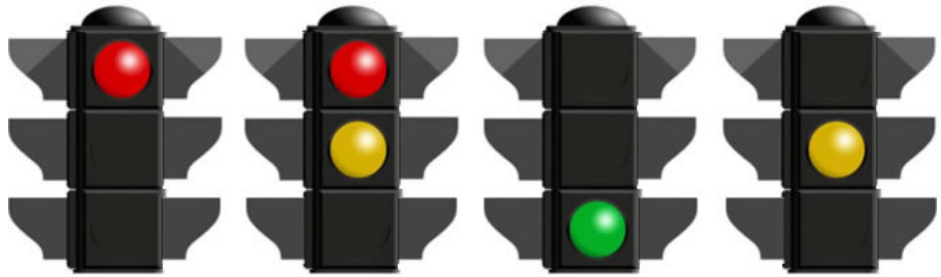
pred esRojoAmarillo(v, a, r: Bool) { $v = \textbf{false} \land a = \textbf{true} \land r = \textbf{true}$ }

Estado válido

```
pred esValido(v, a, r: Bool) {
  esRojo(v, a, r)
    ∨ esRojoAmarillo(v, a, r)
    ∨ esVerde(v, a, r)
    ∨ esAmarillo(v, a, r)
}
```

Ejemplo: Especificando un semáforo

- Sea la siguiente especificación de un semáforo:



- proc *avanzar*(inout v, a, r : Bool) {

  Pre {

  $v = V_0 \land a = A_0 \land r = R_0 \land esValido(V_0, A_0, R_0)$

  }

  Post {

  $(esRojo(V_0, A_0, R_0) \rightarrow esRojoAmarillo(v, a, r))$

  $\land\ (esRojoAmarillo(V_0, A_0, R_0) \rightarrow esVerde(v, a, r)$

  $\land\ (esVerde(V_0, A_0, R_0) \rightarrow esAmarillo(v, a, r))$

  $\land\ (esAmarillo(V_0, A_0, R_0) \rightarrow esRojo(v, a, r))\}$

}

Ejemplo: Semáforo

- Programa a testear:
  - `void avanzar(bool &v, bool &a, bool &r)`
- Test Suite:
  - Test Case #1 (AvanzarRojo):
    - Entrada: ($v = \mathit{false}, a = \mathit{false}, r = \mathit{true}$)
    - Salida Esperada: $v = \mathit{false}, a = \mathit{true}, r = \mathit{true}$
  - Test Case #2 (AvanzarRojoYAmarillo):
    - Entrada: ($v = \mathit{false}, a = \mathit{true}, r = \mathit{true}$)
    - Salida Esperada: $v = \mathit{true}, a = \mathit{false}, r = \mathit{false}$
  - Test Case #3 (AvanzarVerde):
    - Entrada: ($v = \mathit{true}, a = \mathit{false}, r = \mathit{false}$)
    - Salida Esperada: $v = \mathit{false}, a = \mathit{true}, r = \mathit{false}$
  - Test Case #4 (AvanzarAmarillo):
    - Entrada: ($v = \mathit{false}, a = \mathit{true}, r = \mathit{false}$)
    - Salida Esperada: $v = \mathit{false}, a = \mathit{false}, r = \mathit{true}$
- ¿Hay que probar con una configuración no válida?¡No, porque no cumple la Pre!

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

proc *valorAbsoluto*(in $n : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {*True*}
  Post {*result* = $\|n\|$}
}

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con -1, chequear que result=1
- Probar valorAbsoluto con 1, chequear que result=1
- Probar valorAbsoluto con -2, chequear que result=2
- Probar valorAbsoluto con 2, chequear que result=2
- ...etc.
- ¿Cuántas entradas tengo que probar?

Probando (Testeando) programas

- Si los enteros se representan con 32 bits, necesitaríamos probar 2^{32} casos de test.
- Necesito escribir un test suite de 4,294,967,296 test cases.
- Incluso si lo escribo automáticamente, cada test tarda 1 milisegundo, necesitaríamos $1193,04$ horas (49 días) para ejecutar el test suite.
- Cuanto más complicada la entrada (ej: secuencias), más tiempo lleva hacer testing.
- La mayoría de las veces, el testing exhaustivo no es práctico.

Limitaciones del testing

- Al no ser exhaustivo, el testing NO puede probar (demostrar) que el software funciona correctamente.

"El testing puede demostrar la presencia de errores nunca su ausencia" (Dijkstra)

- Una de las mayores dificultades es encontrar un conjunto de tests adecuado:
  - **Suficientemente grande** para abarcar el dominio y maximizar la probabilidad de encontrar errores.
  - **Suficientemente pequeño** para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing.

¿Con qué datos probar?

- **Intuición:** hay inputs que son "parecidos entre sí" (por el tratamiento que reciben)
- Entonces probar el programa con uno de estos inputs, ¿equivaldría a probarlo con cualquier otro de estos parecidos entre sí?
- Esto es la base de la mayor parte de las técnicas
- ¿Cómo definimos cuándo dos inputs son "parecidos"?
  - Si únicamente disponemos de la especificación, nos valemos de nuestra *experiencia*

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

proc $valorAbsoluto(\text{in } n : \mathbb{Z}, \text{out } result : \mathbb{Z})\{$

  Pre $\{True\}$

  Post $\{result = \|n\|\}$

$\}$

Ejemplo:

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con un valor negativo x, chequear que result=-x
- Probar valorAbsoluto con un valor positivo x, chequear que result=x

Ejemplo: valorAbsoluto

- Programa a testear:
  - `int valorAbsoluto(int x)`
- Test Suite:
  - Test Case #1 (cero):
    - Entrada: $(x = 0)$
    - Salida Esperada: $result = 0$
  - Test Case #2 (positivos):
    - Entrada: $(x = 1)$
    - Salida Esperada: $result = 1$
  - Test Case #3 (negativos):
    - Entrada: $(x = -1)$
    - Salida Esperada: $result = 1$

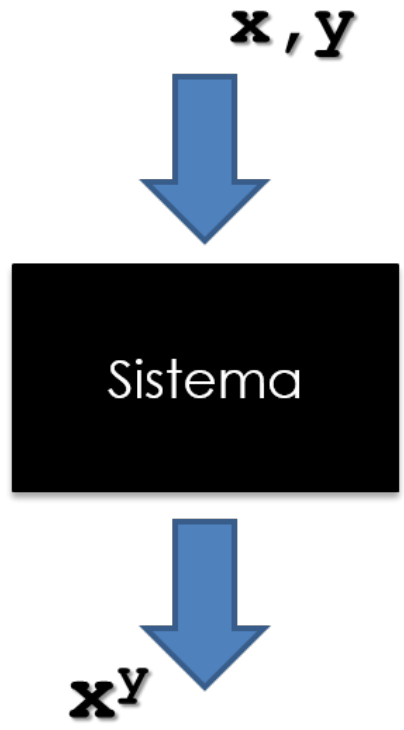
Retomando... ¿Qué casos de test elegir?

1. No hay un algoritmo que proponga casos tales que encuentren todos los errores en cualquier programa.
2. Ninguna técnica puede ser efectiva para detectar todos los errores en un programa arbitrario
3. En ese contexto, veremos dos tipos de criterios para seleccionar datos de test:
   - **Test de Caja Negra**: los casos de test se generan analizando la especificación sin considerar la implementación.
   - **Test de Caja Blanca**: los casos de test se generan analizando la implementación para determinar los casos de test.

Criterios de ***caja negra*** o funcionales

- Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

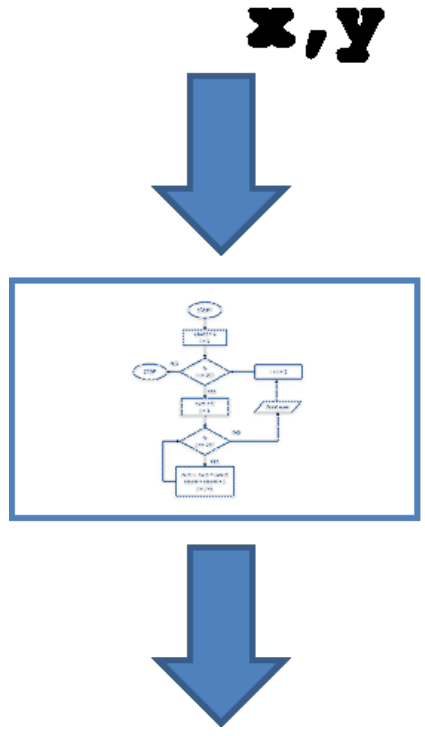
proc *fastexp*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre $\{(x \neq 0 \vee y \neq 0) \wedge (y \geq 0)\}$
  Post $\{result = x^y)\}$
}



Criterios de ***caja blanca*** o estructurales

- Los datos de test se derivan a partir de la estructura interna del programa.

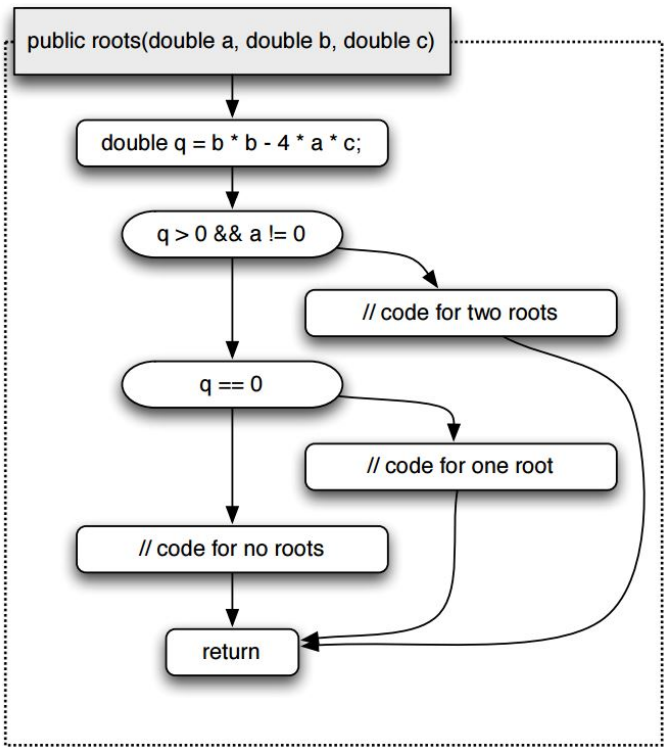
```
int fastexp(int x, int y) {
  int z = 1;
  while( y != 0 ) {
    if(impar(y)) then {
      z = z * x;
      y = y - 1;
    }
    x = x * x;
    y = y / 2;
  }
  return z;
}
```



¿Qué pasa si y es potencia de 2?

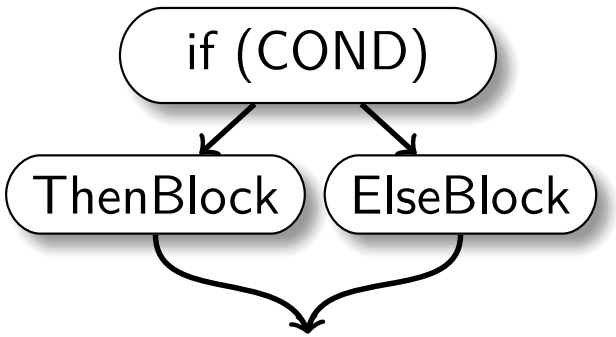
¿Qué pasa si $y = 2^n - 1$?

Control-Flow Graph

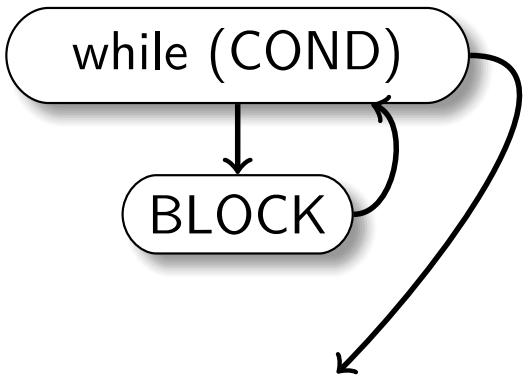


- El control flow graph (CFG) de un programa es sólo una representación gráfica del programa.
- El CFG es independiente de las entradas (su definición es estática)
- Se usa (entre otras cosas) para definir criterios de adecuación para test suites.
- Cuanto más *partes* son ejercitadas (cubiertas), mayores las chances de un test de descubrir una falla
- *partes* pueden ser: nodos, arcos, caminos, decisiones...

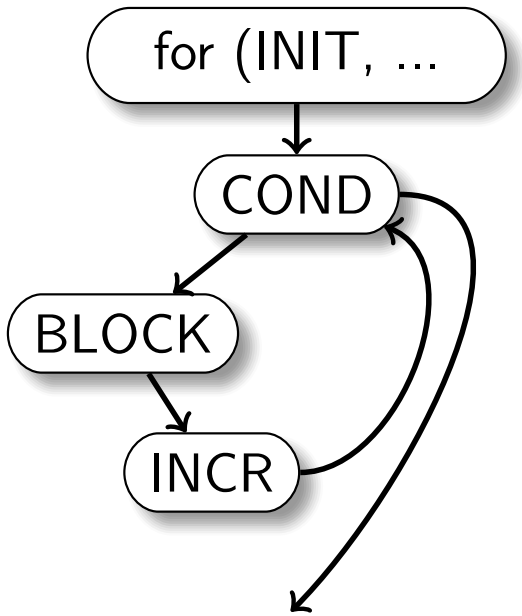
Control Flow Patterns




```
if (COND)
    ThenBlock
else
    ElseBlock
```




```
while (COND)
    BLOCK
```




```
for (INIT; COND; INCR)
    BLOCK
```

Ejemplo #1: valorAbsoluto

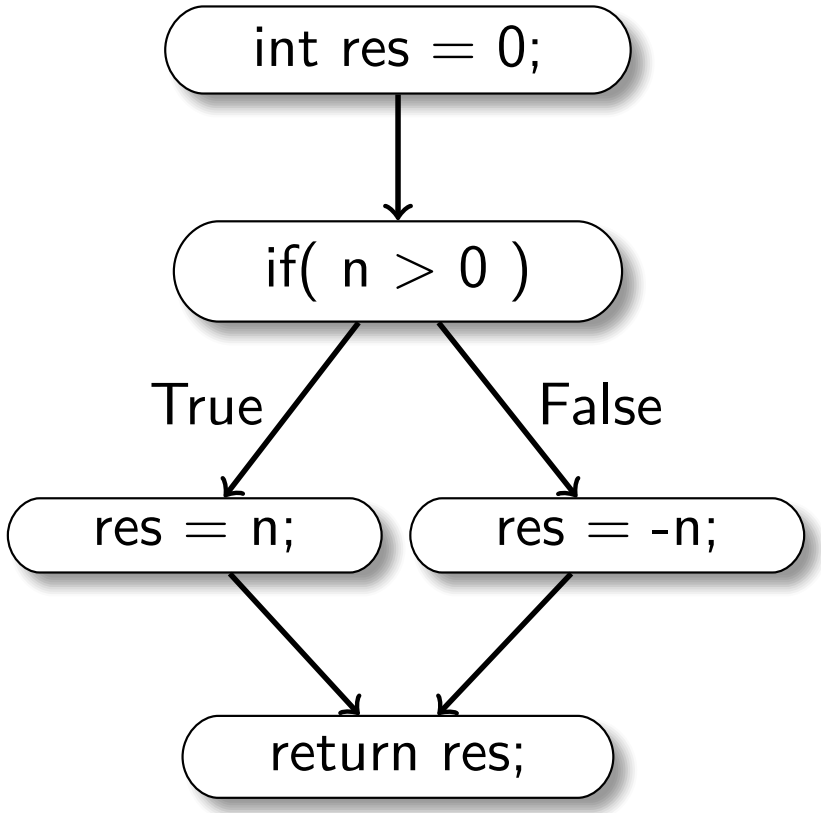
proc *valorAbsoluto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {*True*}
  Post {*result* = $\|x\|$}
}

```
int valorAbsoluto(int n) {
  int res = 0;
  if( n > 0 ) {
    res = n;
  } else {
    res = -n;
  }
  return res;
}
```

Ejemplo #1: valorAbsoluto

Control Flow Graph

```
int valorAbsoluto(int n) {
  int res = 0;
  if( n > 0 )  {
    res = n;
  } else {
    res = -n;
  }
  return res;
}
```



Ejemplo #2: Sumar

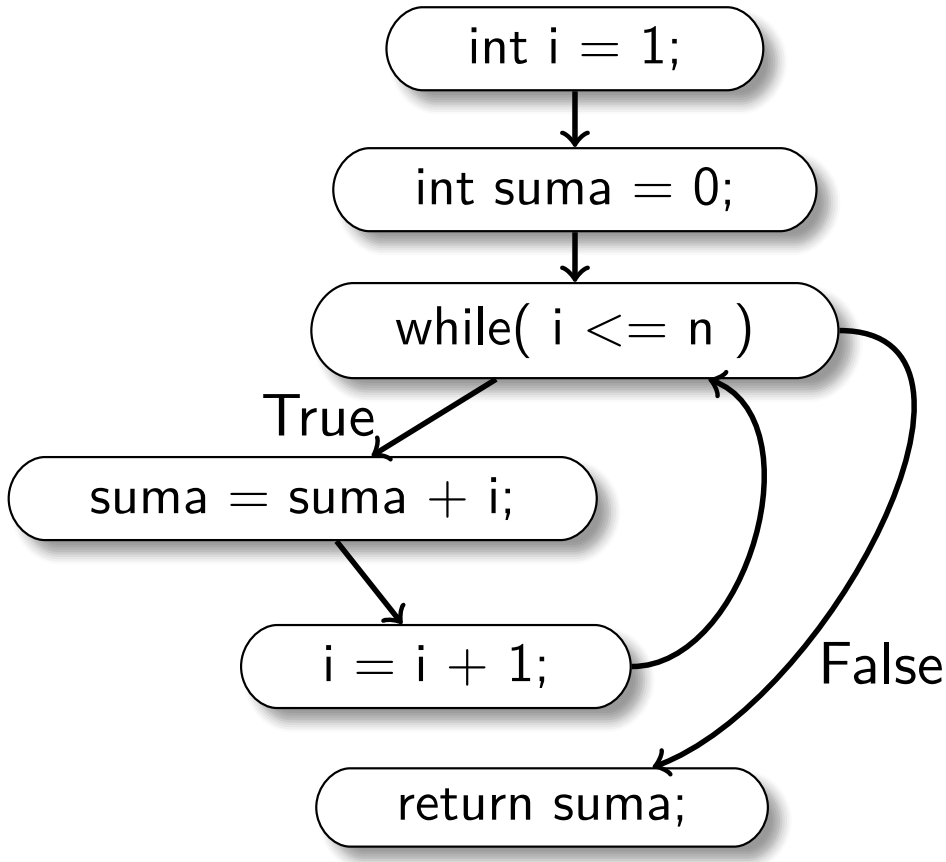
proc *sumar*(in $n : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre $\{n \geq 0\}$
  Post $\{result = \sum_{i=1}^{n} i\}$
}

```
int sumar(int n) {
  int i = 1;
  int suma = 0;

  while( i ≤ n ) {
    suma = suma + i ;
    i = i + 1;
  }
  return suma;
}
```

Ejemplo #2: Sumar

Control Flow Graph

```
int sumar(int n) {
  int i = 1;
  int suma = 0;

  while( i <= n ) {
    suma = suma + i;
    i = i + 1;
  }
  return suma;
}
```



Ejemplo #3: crearVectorN

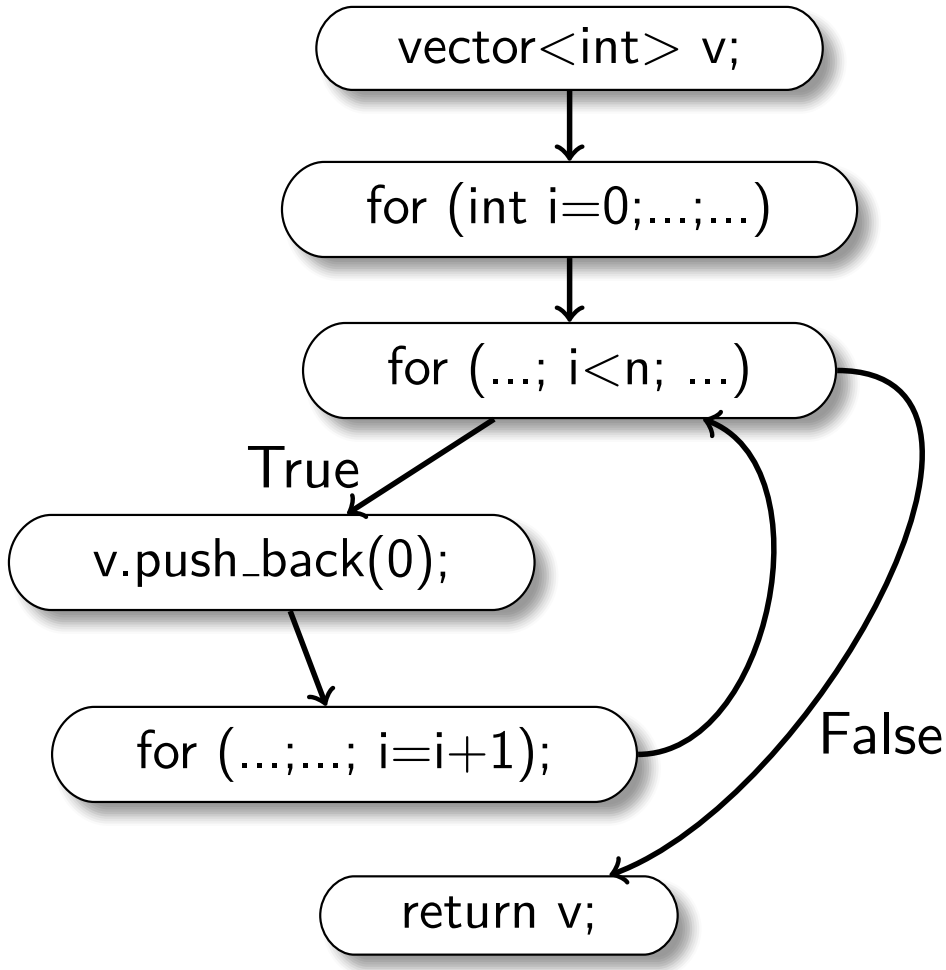
proc *crearVectorN*(in $n : \mathbb{Z}$, out *result* : $seq\langle\mathbb{Z}\rangle$){
    Pre $\{n \geq 0\}$
    Post $\{|result| = n \wedge \#apariciones(result, 0) = n\}$
}

```
vector<int> crearVectorN(int n) {
  vector<int> v;
  for (int i=0; i<n; i=i+1) {
    v.push_back(0);
  }
  return v;
}
```

Ejemplo #3: crearVectorN

Control Flow Graph

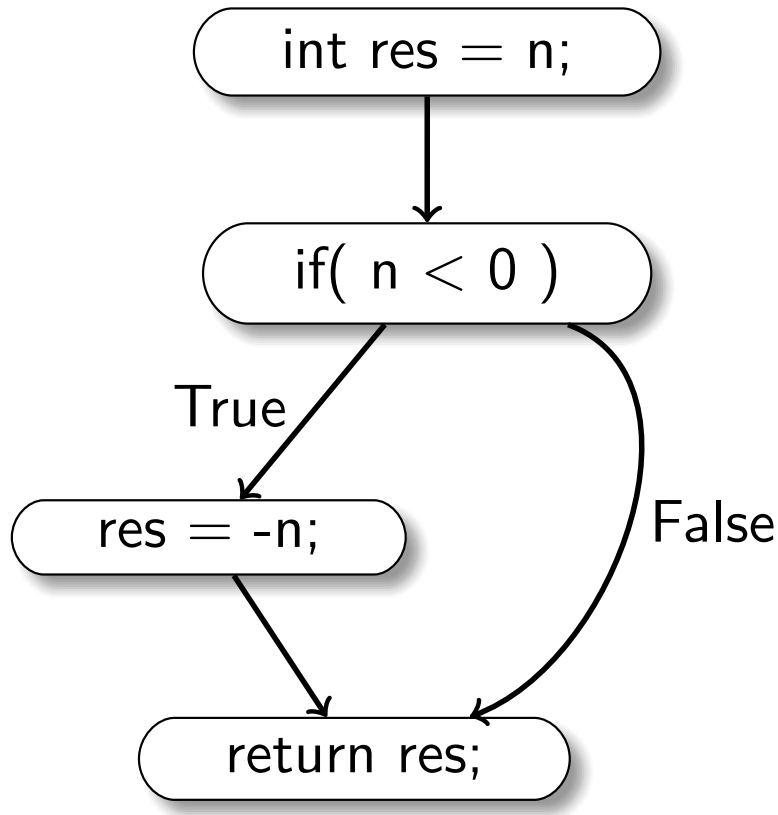
```
vector<int> crearVectorN(int n) {
  vector<int> v;
  for (int i=0; i<n; i=i+1) {
    v.push_back(0);
  }
  return v;
}
```



Ejemplo #4: valorAbsoluto

Control Flow Graph

```
int valorAbsoluto(int n) {
   int res = n;
   if( n < 0 )  {
      res = -n;
   }
   return res;
}
```



Criterios de Adecuación

- ¿Cómo sabemos que un *test suite* es *suficientemente bueno*?
- Un criterio de adecuación de test es un predicado que toma un valor de verdad para una tupla <*programa*, *test suite*>
- Usualmente expresado en forma de una regla del estilo:

  todas las sentencias deben ser ejecutadas

Cubrimiento de Sentencias

- Criterio de Adecuación: cada nodo (sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case
- Idea: un defecto en una sentencia sólo puede ser revelado ejecutando el defecto
- Cobertura:

$$\frac{cantidad\ nodos\ ejercitados}{cantidad\ nodos}$$

Cubrimiento de Arcos

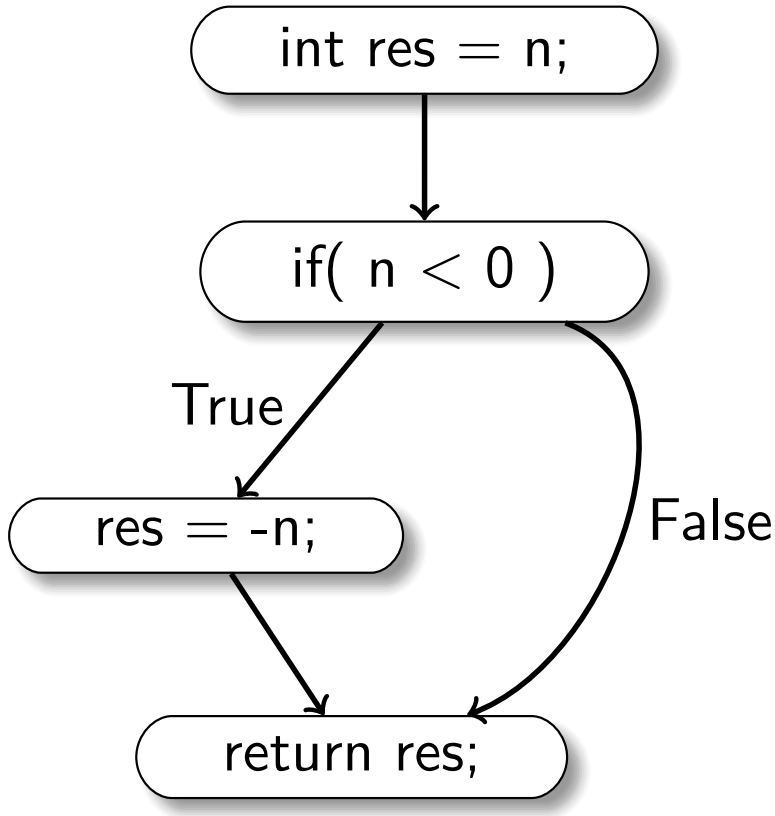
- Criterio de Adecuación: todo arco en el CFG debe ser ejecutado al menos una vez por algún test case
- Si recorremos todos los arcos, entonces recorremos todos los nodos. Por lo tanto, el cubrimiento de arcos incluye al cubrimiento de sentencias.
- Cobertura:

$$\frac{cantidad\ arcos\ ejercitados}{cantidad\ arcos}$$

- El cubrimiento de sentencias (nodos) no incluye al cubrimiento de arcos. ¿Por qué?

Cubrimiento de Nodos no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los nodos pero que no cubra todos los arcos.

Cubrimiento de Decisiones (o Branches)

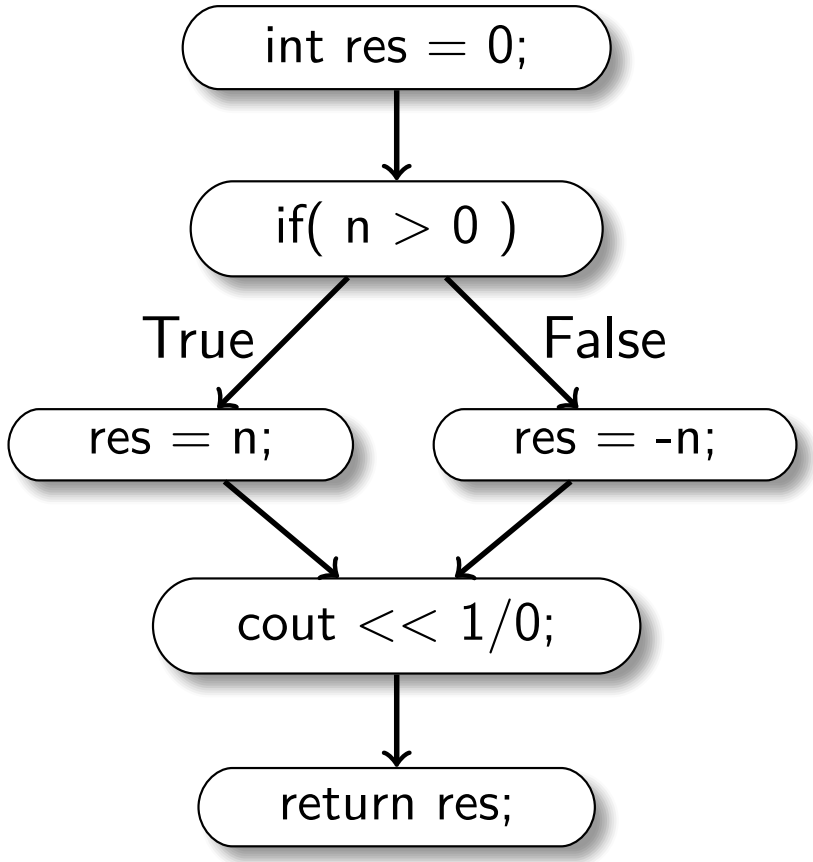
- Criterio de Adecuación: cada decisión (arco True o arco False) en el CFG debe ser ejecutado
- Por cada arco **True** o arco **False**, debe haber al menos un test case que lo ejercite.
- Cobertura:

$$\frac{cantidad\ decisiones\ ejercitadas}{cantidad\ decisiones}$$

- El cubrimiento de decisiones **no implica** el cubrimiento de los arcos del CFG. ¿Por qué?

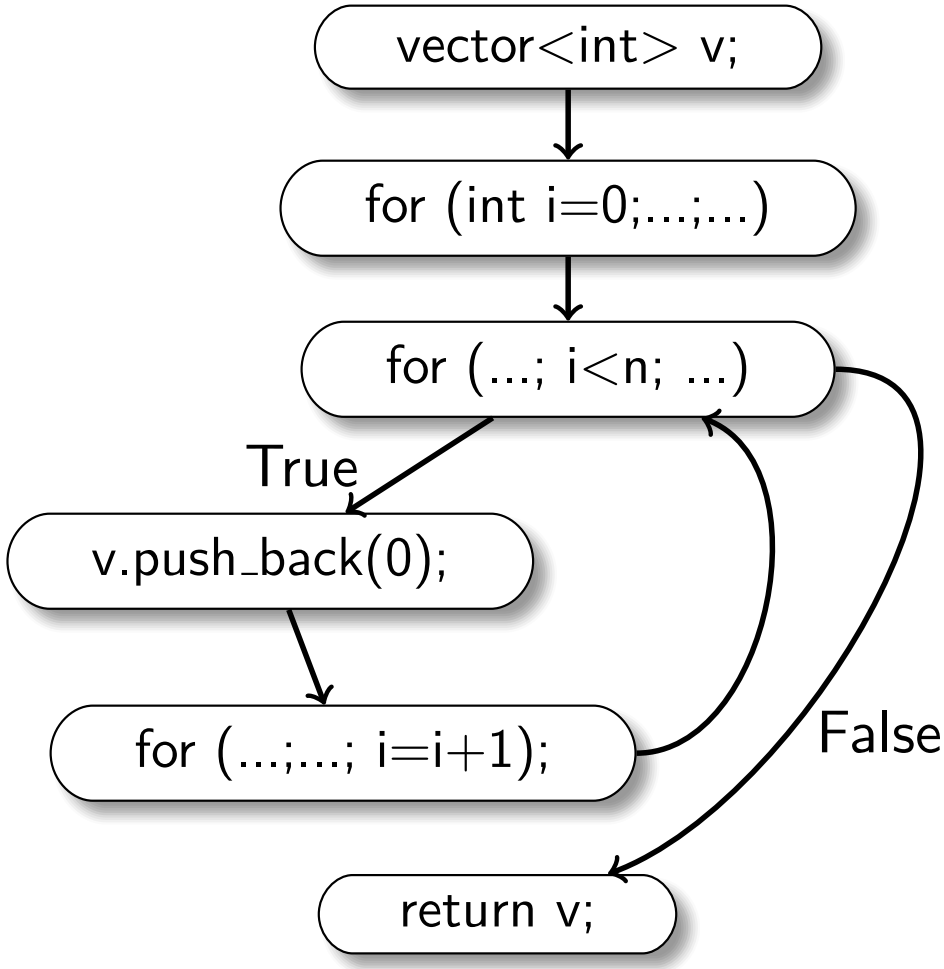
Cubrimiento de Branches no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los branches pero que no cubra todos los arcos.

CFG de crearVectorN



- ¿Cuántos nodos (sentencias) hay? 6
- ¿Cuántos arcos (flechas) hay? 6
- ¿Cuántas decisiones (arcos True y arcos False) hay? 2

Cubrimiento de Condiciones Básicas

- Una condición básica es una fórmula atómica (i.e. no divisible) que componen una decisión.
  - Ejemplo: `(digitHigh==1 || digitLow==-1) && len>0`
  - Condiciones básicas:
    - `digitHigh==1`
    - `digitLow==-1`
    - `len>0`
  - No es condición básica: `(digitHigh==1 || digitLow==-1)`
- Criterio de Adecuación: cada condición básica de cada decisión en el CFG debe ser evaluada a verdadero y a falso al menos una vez
- Cobertura:

$$\frac{cantidad\ de\ valores\ evaluados\ en\ cada\ condicion}{2 * cantidad\ condiciones\ basicas}$$

Cubrimiento de Condiciones Básicas

- Sea una única decisión:
  `(digitHigh==1 || digitLow==-1) && len>0`
- Y el siguiente test case:

Entrada	digitHigh==1?	digitLow==-1?	len>0?
digitHigh=1, digitLow=0 len=1,	True	False	True

- ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{3}{2 * 3} = \frac{3}{6} = 50\,\%$$

Cubrimiento de Condiciones Básicas

- Sea una única decisión:
  `(digitHigh==1 || digitLow==-1) && len>0`
- Y el siguiente test case:

Entrada	digitHigh==1?	digitLow==-1?	len>0?
digitHigh=1, digitLow=0 len=1,	True	False	True
digitHigh=0, digitLow=-1 len=0,	False	True	False

- ¿Cuál es el cubrimiento de condiciones básicas?

$$C_{\text{cond.básicas}} = \frac{6}{2 * 3} = \frac{6}{6} = 100\,\%$$

Cubrimiento de Branches y Condiciones Básicas

- **Observación** Branch coverage no implica cubrimiento de Condiciones Básicas
  - Ejemplo: ***if(a && b)***
  - Un test suite que ejercita solo $a = true, b = true$ y $a = false, b = true$ logra cubrir ambos branches de ***if(a && b)***
  - Pero: no alcanza cubrimiento de decisiones básica ya que falta $b = false$
- El criterio de cubrimiento de Branches y condiciones básicas necesita 100 % de cobertura de branches y 100 % de cobertura de condiciones básicas
- Para ser aprobado, todo software que controla un avión necesita ser testeado con cubrimiento de branches y condiciones básicas (RTCA/DO-178B en EEUU y EUROCAE ED-12B en UE).

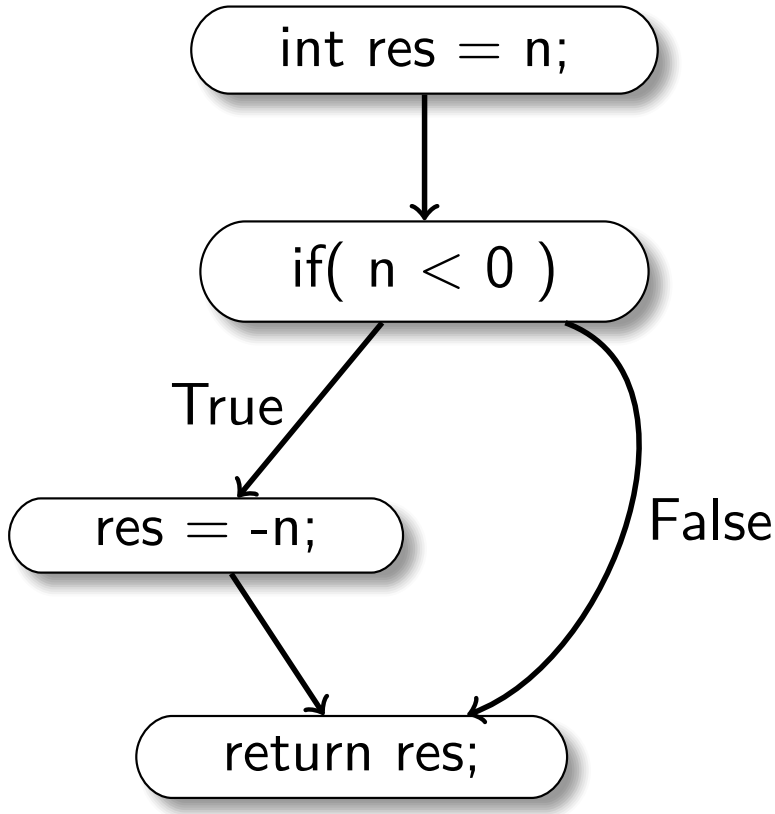
Cubrimiento de Caminos

- Criterio de Adecuación: cada camino en el CFG debe ser transitado por al menos un test case
- Cobertura:

$$\frac{cantidad\ caminos\ transitados}{cantidad\ total\ de\ caminos}$$

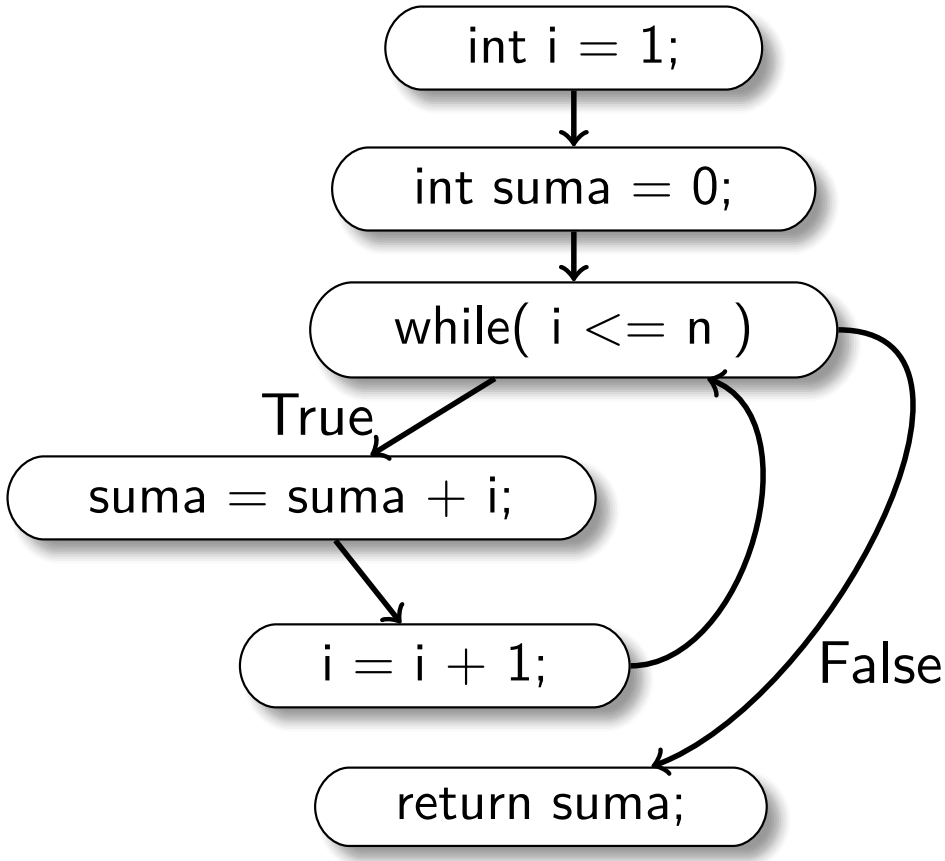
Caminos para el CFG de valorAbsoluto

Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? 2

Caminos para el CFG de sumar



¿Cuántos caminos hay en este CFG?

La cantidad de caminos no está acotada (∞)

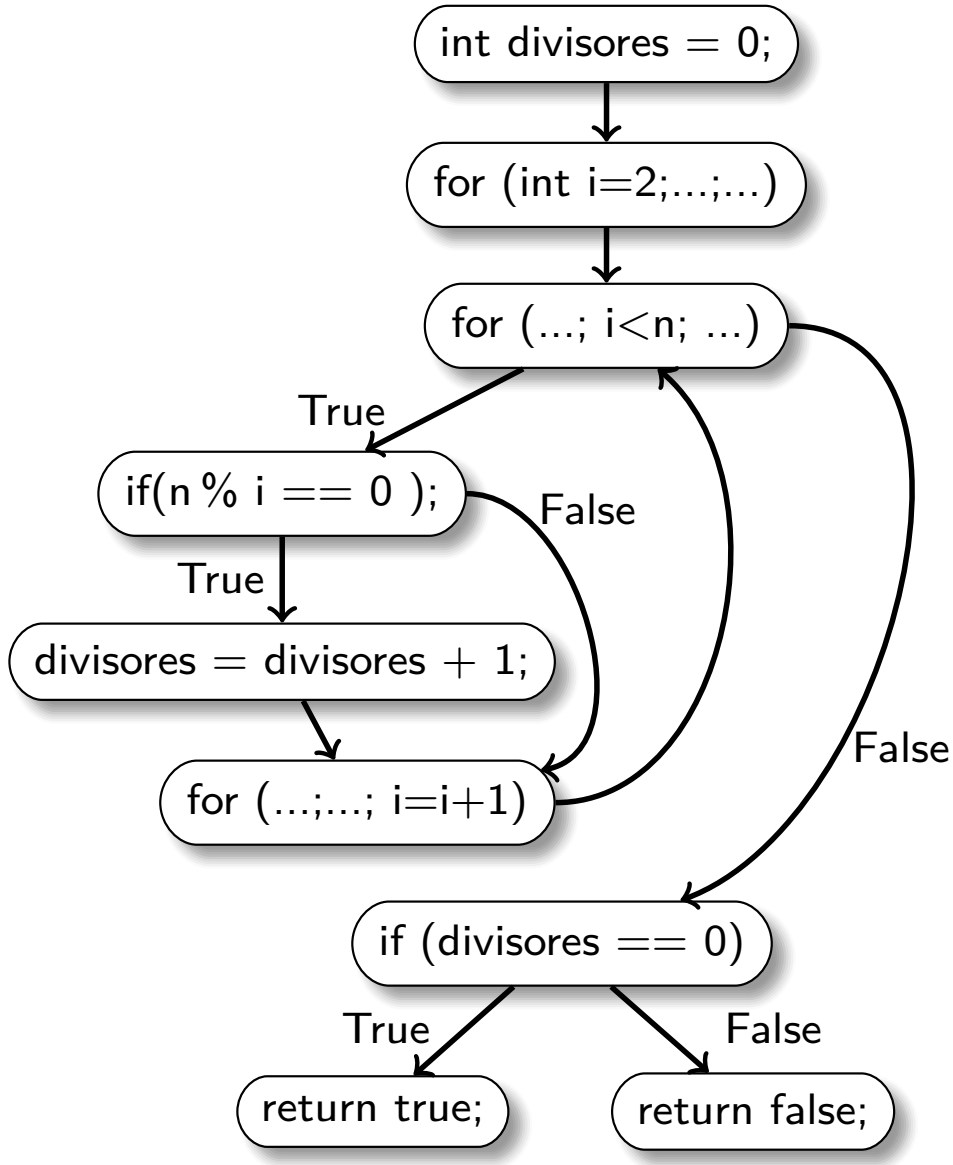
Resumen: Criterios de Adecuación Estructurales

- En todos estos criterios se usa el CFG para obtener una métrica del test suite
- **Sentencias**: cubrir todas los nodos del CFG
- **Arcos**: cubrir todos los arcos del CFG
- **Decisiones (Branches)**: Por cada `if`, `while`, `for`, etc., la guarda fue evaluada a verdadero y a falso.
- **Condiciones Básicas**: Por cada componente básico de una guarda, este fue evaluado a verdadero y a falso.
- **Caminos**: cubrir todos los caminos del CFG. Como no está acotado o es muy grande, se usa muy poco en la práctica.

esPrimo()

Sea el siguiente programa que decide si un número $n > 1$ es primo:

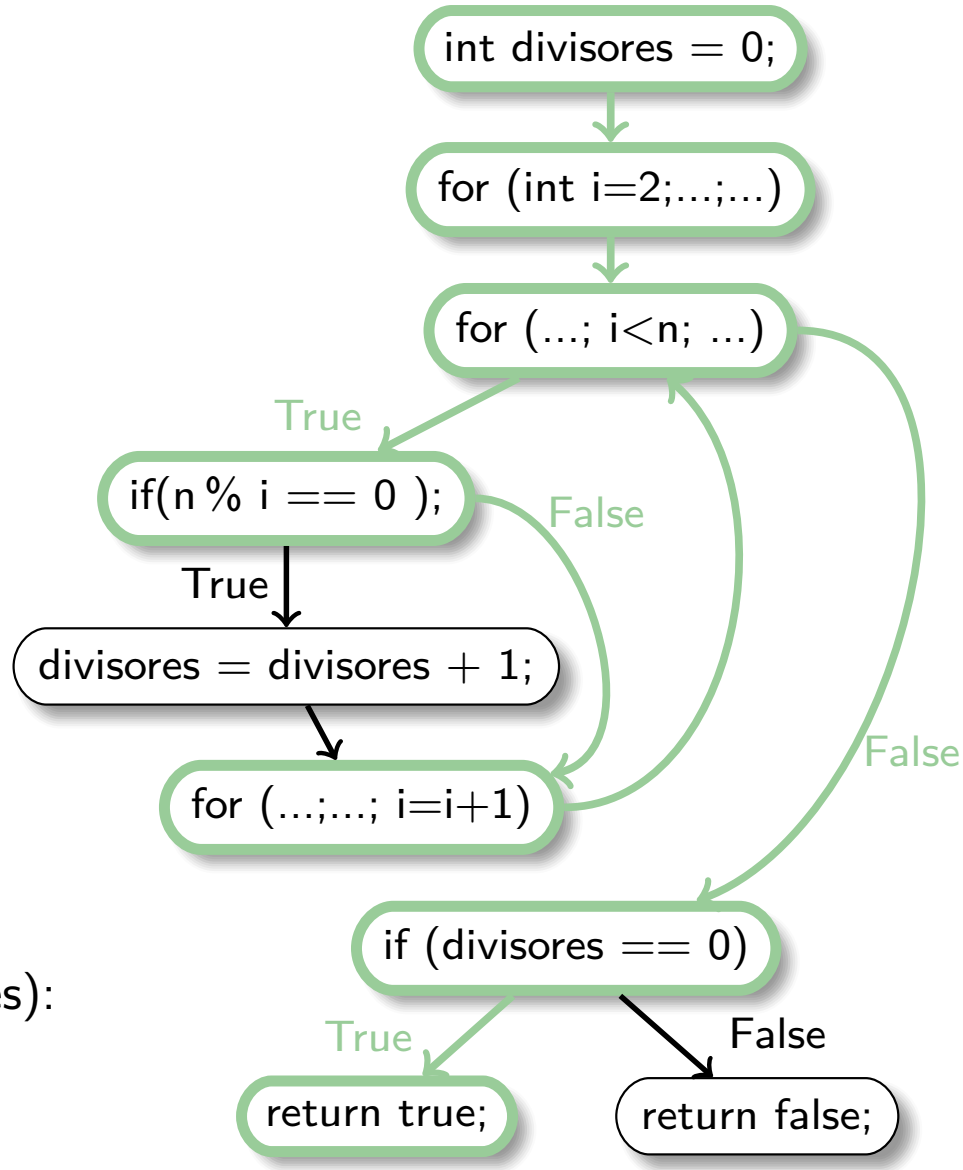
```
bool esPrimo(int n) {
  int divisores = 0;
  for(int i=2; i<n; i=i+1) {
    if( n % i == 0 ) {
      divisores = divisores + 1;
    }
  }
  if (divisores == 0) {
    return true;
  } else {
    return false;
  }
}
```



esPrimo()

Test Suite #1: usando 2 test case, una entrada par y una entrada impar

- Test Case #1: valorPar
  - Entrada: $n = 2$
  - Salida esperada: $result = true$
- Test Case #2: valorImpar
  - Entrada: $n = 3$
  - Salida esperada: $result = true$



- Cubrimientos del test suite:
  - Cubrimiento de sentencias:

$$Cov_{sentencias} = \frac{7}{9} \sim 77\,\%$$

  - Cubrimiento de decisiones (branches):

$$Cov_{branches} = \frac{4}{6} \sim 66\,\%$$

esPrimo()

Test Suite #2: usando 2 test case, un primo como entrada y uno no primo

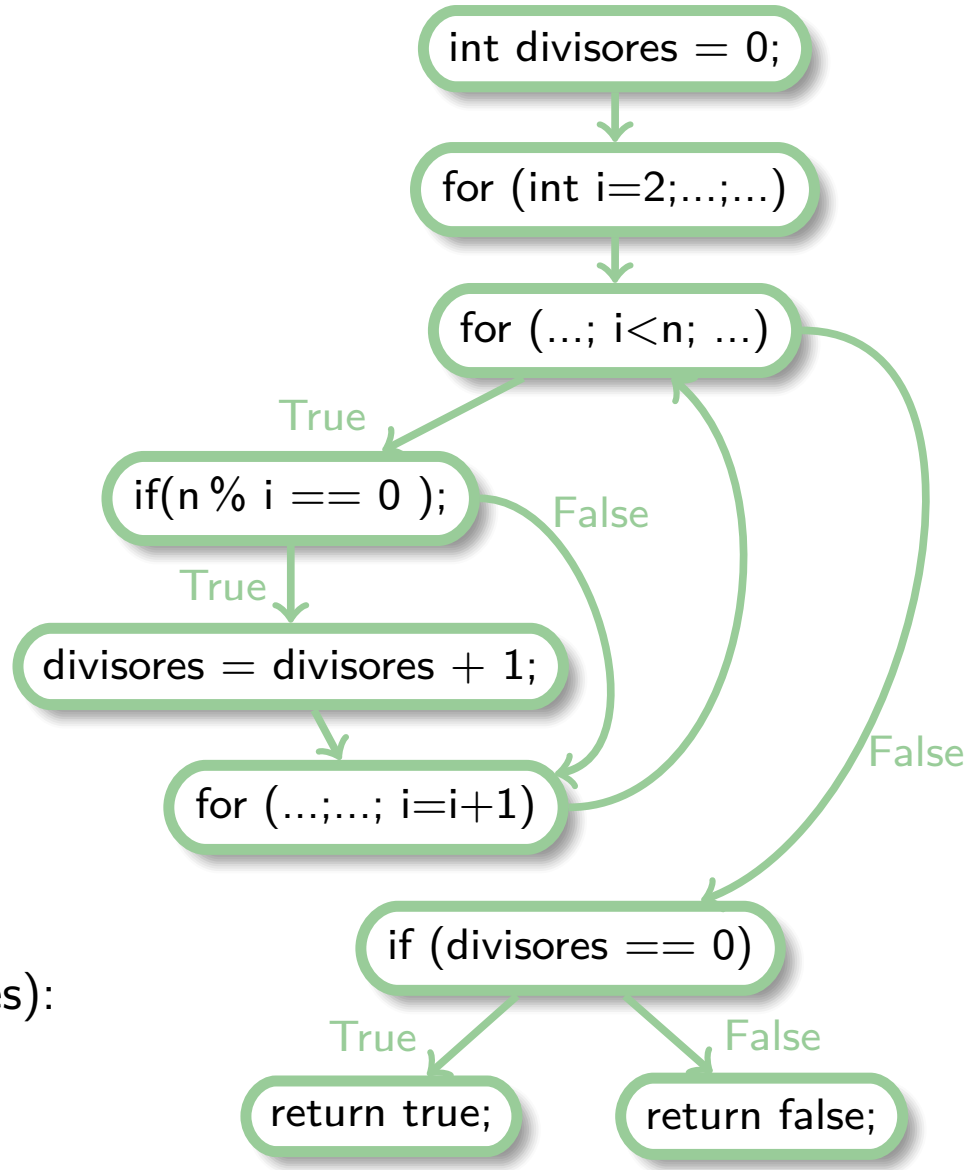
- Test Case #1: valorPrimo
  - Entrada: $n = 3$
  - Salida esperada: $result = true$
- Test Case #2: valorNoPrimo
  - Entrada: $n = 4$
  - Salida esperada: $result = false$

- Cubrimientos del test suite:
  - Cubrimiento de sentencias:

$$Cov_{sentencias} = \frac{9}{9} \sim 100\,\%$$

  - Cubrimiento de decisiones (branches):

$$Cov_{branches} = \frac{6}{6} \sim 100\,\%$$



Ordenar una secuencia

proc ordenar(inout a $seq\langle Z \rangle$) {

Pre { $a = A_0$ }

Post { $(\forall i, j : \mathbb{Z})(0 \leq i \leq j < |a|) \rightarrow_L a[i] \leq a[j]) \wedge$

$(\forall e : T)(\#apariciones(a, e) = \#apariciones(A_0, e))$}

}

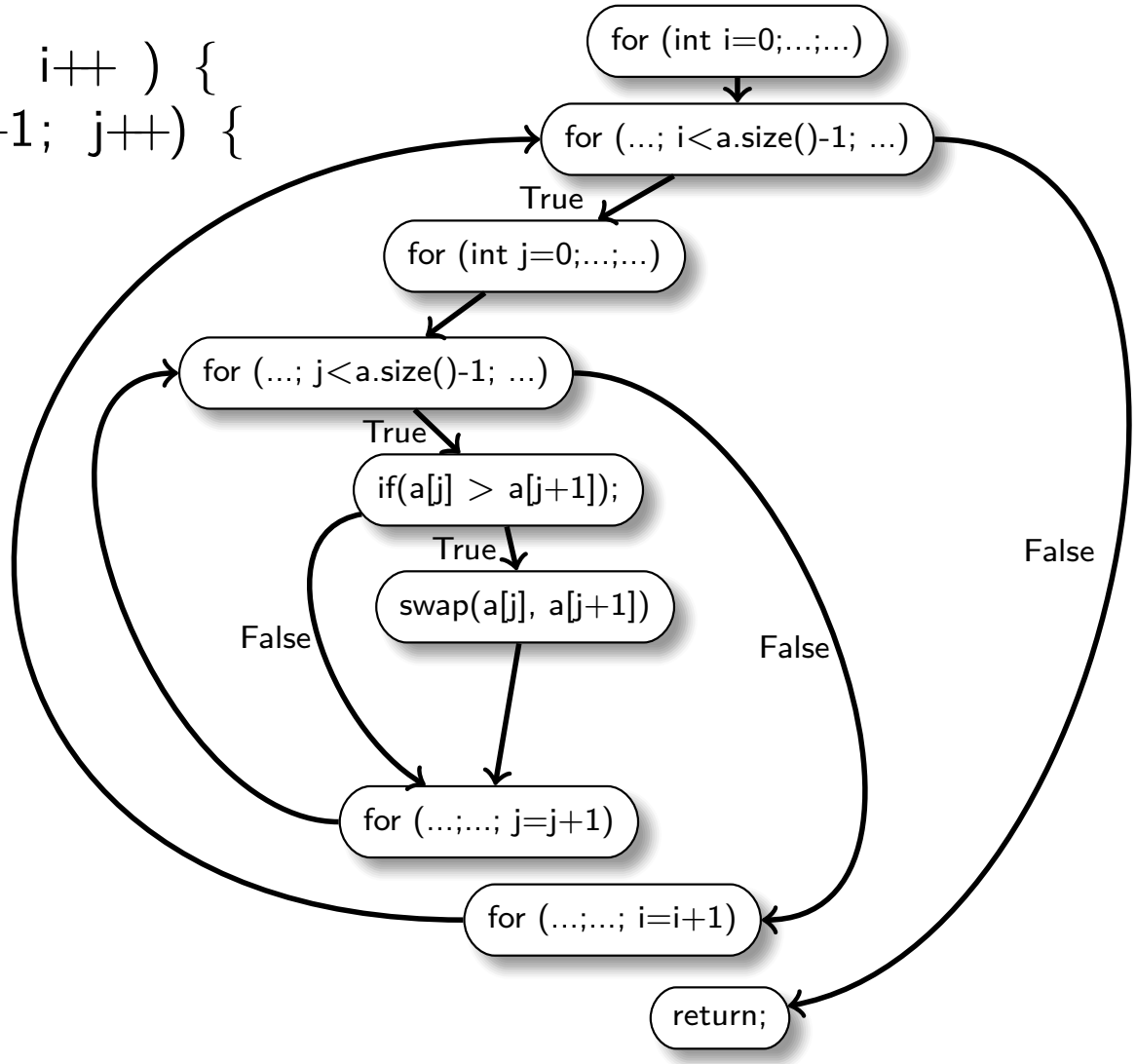
```
for(int i = 0; i<a.size()−1; i++ ) {
  for(int j = 0; j<a.size()−1; j++) {
    if(a[j]>a[j+1]) {
      swap(a[j], a[j+1]);
    }
  }
}
return;
```

Ver este algoritmo en acción en:
`https://visualgo.net/es/sorting` (ORDENAMIENTO BURBUJA)

Ordenar una secuencia

Bubble Sort

```
for(int i = 0; i<a.size()-1; i++ ) {
  for(int j = 0; j<a.size()-1; j++) {
    if(a[j]>a[j+1]) {
      swap(a[j], a[j+1]);
    }
  }
}
return;
```



Cubrimientos

Con los siguiente test case se arman distintos Test Suite:

- Test Case A: secuencia vacía
  - Entrada: $\langle\rangle$ – Salida esperada: $\langle\rangle$
- Test Case B: secuencia con un único elemento
  - Entrada: $\langle 0\rangle$ – Salida esperada: $\langle 0\rangle$
- Test Case C: secuencia ordenada con mas de un elemento
  - Entrada: $\langle 1, 2\rangle$ – Salida esperada: $\langle 1, 2\rangle$
- Test Case D: secuencia desordenada con dos elementos
  - Entrada: $\langle 2, 1\rangle$ – Salida esperada: $\langle 1, 2\rangle$

Calcular el cubrimiento de sentencias (nodos) y de decisiones (branches) de los Test Suite:

- Test Suite I: Case A
- Test Suite II: Case A, B
- Test Suite III: Case B
- Test Suite IV: Case B, D
- Test Suite V: Case A, B, C, D

Discusión

- ¿Puede haber partes (nodos, arcos, branches) del programa que no sean alcanzables con **ninguna** entrada válida (i.e. que cumplan la precondición)?
- ¿Qué pasa en esos casos con las métricas de cubrimiento?
- Existen esos casos (por ejemplo: código defensivo o código que sólo se activa ante la presencia de un estado inválido)
- El 100 % de cubrimiento suele ser no factible, por eso es una medida para analizar con cuidad y estimar en función al proyecto (ejemplo: 70 %, 80 %, etc.)

Edsger Dijkstra

"El testing puede demostrar la presencia de errores nunca su ausencia"

Bibliografía

- David Gries - The Science of Programming
  - Chapter 22 - Notes on Documentation
    - Chapter 22.1 - Indentation
    - Chapter 22.2 - Definitions and Declarations of Variables
- Pezze, Young - Software Testing and Analysis
  - Chapter 1 - Software Test and Analysis in a Nutshell
  - Chapter 12 - Structural Testing