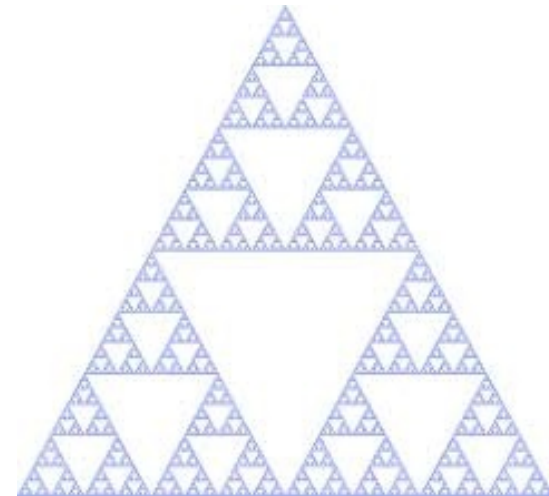
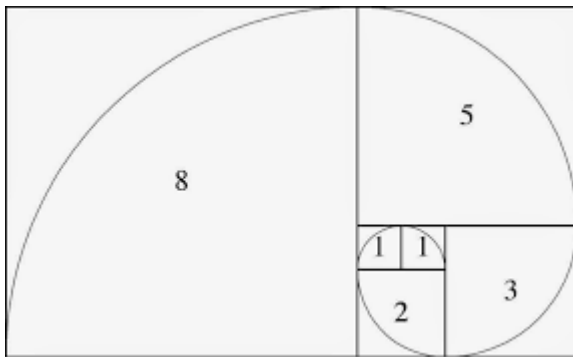


RECURSION

Técnica utilizada en programación en la cual un método pueden llamarse a sí mismo para resolver determinados tipos de problemas.

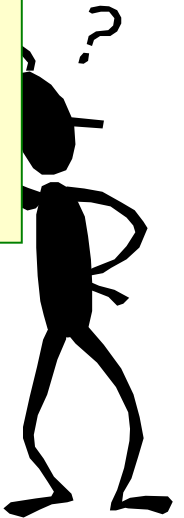


Método cuentaRegresiva

- Contar los números desde el 10 hacia el 1,

```
MODULO cuentaRegresiva1(ENTERO n) RETORNA VACIO
  PARA i=n HASTA 1 PASO -1 HACER
    ESCRIBIR i
  FIN PARA
FIN MODULO
```

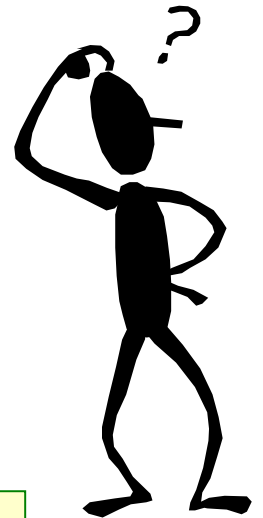
- Probemos hacerlo llamando a sí mismo



Método cuentaRegresiva

```
MODULO cuentaRegresiva1(ENTERO n) RETORNA VACIO
  PARA i=n HASTA 1 PASO -1 HACER
    ESCRIBIR i
  FIN PARA
FIN MODULO
```

```
MODULO cuentaRegresiva2(ENTERO n) RETORNA VACIO
  ESCRIBIR n
  ....?
  cuentaRegresiva2(..)..
FIN MODULO
```



Definición

```
public class Recursividad {  
    ...  
    static public void cuentaReg (int n) {  
        System.out.println(n);  
        cuentaReg (n-1);  
    }  
  
    public static void main(String[] args) {  
        cuentaReg(10);  
    }  
    ...  
}
```

- Llamada infinita,
para evitarlo....

MODULO cuentaReg(ENTERO numero) RETORNA vacio
SI numero > 0 ENTONCES
 ESCRIBE numero
 cuentaRegresiva(numero - 1);
FIN SI
FIN MODULO

Diseño, ejemplo

$$n! = \begin{cases} 1 & \text{si } n=0 \\ n (n-1) (n-2) \dots * 1 & \text{si } n > 0 \end{cases}$$

$$1! = 1$$

$$2! = 1! * 2$$

$$\begin{aligned} 3! &= 2! * 3 \\ &= (1! * 2) * 3 \\ &= 1 * 2 * 3 \end{aligned}$$

$$n! = \begin{cases} 1 & (\text{para } n = 0) \\ (n-1)! * n & (\text{para } n > 0) \end{cases}$$

Diseño recursivo

- Descomposicion
- Composicion
- Caso base

Ejemplo

$$n! = \begin{cases} 1 & \text{si } n=0 \\ n * (n-1) * \dots * 1 = n * (n-1)! & \text{si } n > 0 \end{cases}$$

Diseño recursivo

- Descomposicion: $(n-1)!$
- Composicion: $* n$
- Caso base: $1!$

MODULO factorial (n)

(* n es entero *)

entero: fact

SI (n > 0)

fact ← n* factorial (n-1)

SINO

fact ← 1

FIN SI

sale fact

FIN MODULO

MODULO factorial (n)

(* n es entero, retorna el factorial de ese numero *)

entero: fact

fact ← 1

SI (n > 1)

fact ← factorial (n-1) * n

FIN SI

sale fact

FIN MODULO

Finalización de algoritmos recursivos

- Caso base: No ejecuta un llamado recursivo
 - **Detiene la recursión.**
- Cada llamado sucesivo a si mismo debe ser una “**version mas pequeña de si mismo**”
- Un algoritmo recursivo debe contener:
 - Un argumento que describe el problema mas pequeño
 - Un caso base que eventualmente sera alcanzado



Un método recursivo debe tener al menos un caso base o de corte.

Diseño de algoritmos recursivos

- Para el diseño se debe considerar 3 partes:

1. Como sería un problema idéntico mas pequeño
Descomposicion

1. Como combinar las soluciones a los problemas mas pequeños para formar la solucion de un problema más grande?
Composicion

3. Cual es el problema mas pequeno que puede resolverse facilmente sin descomposicion?
Caso Base o condicion de corte

Factorial en Java

```
public static int factorial(int n) {  
    int fact;  
  
    if (n > 1)  
        fact = factorial(n - 1) * n;  
    else  
        fact = 1;  
  
    return fact;  
}
```

Caso recursivo

composición

Combinar las soluciones de los problemas mas pequeños formando la solución de un problema más grande

descomposición

Descomponer un problema en más pequeños

caso base

Determinar el problema más pequeño que no puede descomponerse

Visión General

- Recursion general

- Es una aproximación natural para resolver algunos problemas (no todos)
- Un algoritmo recursivo se utiliza a si mismo para resolver uno o mas problemas idénticos

- en Java

- Los metodos recursivos implementan algoritmos recursivos
- Un método recursivo incluye uno (o mas) llamados a sí mismo
- Un método recursivo debe tener uno (o más) caso que termine la ejecución recursiva



Recursión

Conceptos

- **Recursividad** va ligado al concepto de **repetición**.
- Algo es recursivo si **se define en términos de sí mismo** (cuando para definirse hace mención a sí mismo).
- Su contraposición son los algoritmos iterativos, que hacen uso de bucles while, do-while, for, etc.
- Para que una definición recursiva sea válida, la referencia a sí misma debe ser relativamente más sencilla que el caso considerado.

Recursión: definición en términos de sí mismo

Elementos de la Recursión

- **Axioma:**
 - Es un caso donde el problema puede resolverse sin tener que hacer uso de una nueva llamada a sí mismo.
 - Evita la continuación indefinida de las partes recursivas.
- **Formula Recursiva:**
 - Relaciona el resultado del algoritmo con resultados de casos más simples. Se hacen nuevas llamadas a la función, pero están más próximas al caso base.

Ejecución de un método recursivo

- La sentencia **return** es la última en ejecutarse en cada llamada

La sentencia return devuelve el control justo después del punto donde el método recursivo fue invocado



Al volver de la llamada recursiva se sigue ejecutando a partir del punto de donde se hizo la última llamada

- Si el método recursivo devuelve un valor, generalmente se utiliza para:
 - Hacer algún cálculo con el valor recibido
 - Almacenarlo en una variable
 - Utilizarlo para imprimir
- Si el método recursivo no devuelve un valor, seguirá ejecutando la sentencia siguiente a la llamada

Método factorial

```
public static int factorial(int n)
{
    int fact;

    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;

    return fact;
}
```

The diagram illustrates the recursive factorial method with four callouts:

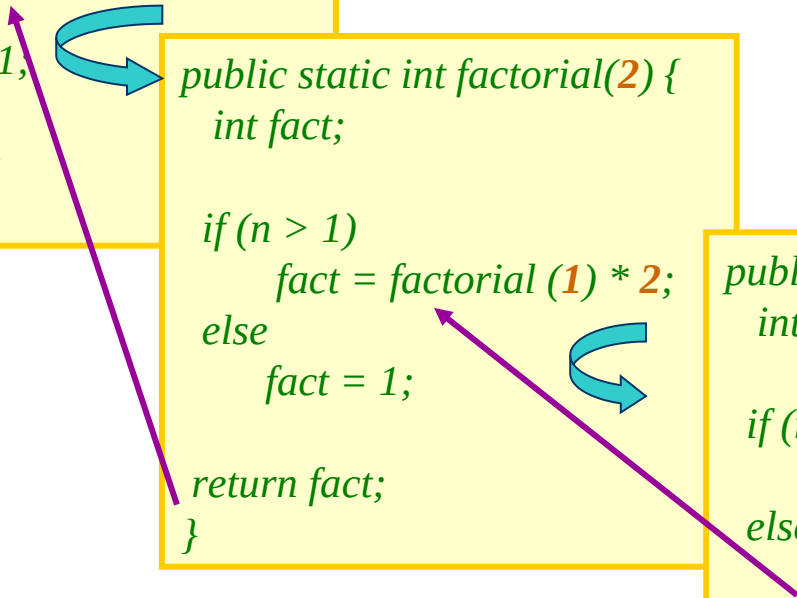
- CASO RECURSIVO** (yellow box) points to the `factorial(n - 1)` part of the recursive call.
- composición** (green box) points to the multiplication operation `*`.
- descomposición** (blue box) points to the subtraction operation `-` in `n - 1`.
- CASO BASE** (orange box) points to the `fact = 1;` line in the `else` block.

Traza de ejecución

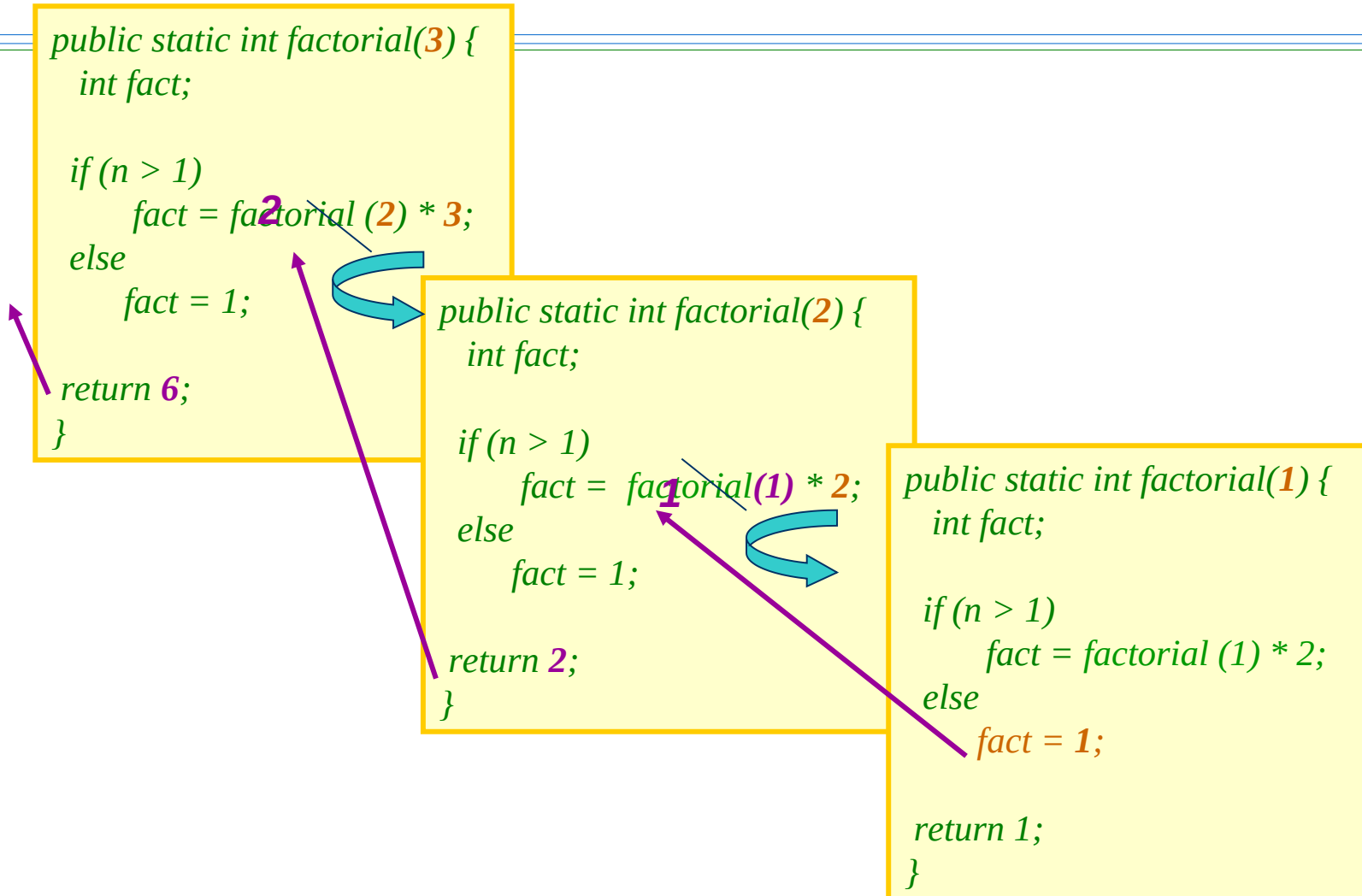
```
public static int factorial(3) {  
    int fact;  
  
    if (n > 1)  
        fact = factorial (2) * 3;  
    else  
        fact = 1;  
  
    return fact;  
}
```

```
public static int factorial(2) {  
    int fact;  
  
    if (n > 1)  
        fact = factorial (1) * 2;  
    else  
        fact = 1;  
  
    return fact;  
}
```

```
public static int factorial(1) {  
    int fact;  
  
    if (n > 1)  
        fact = factorial (1) * 2;  
    else  
        fact = 1;  
  
    return fact;  
}
```

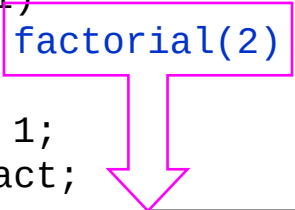


Ejecución de factorial




```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```



```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

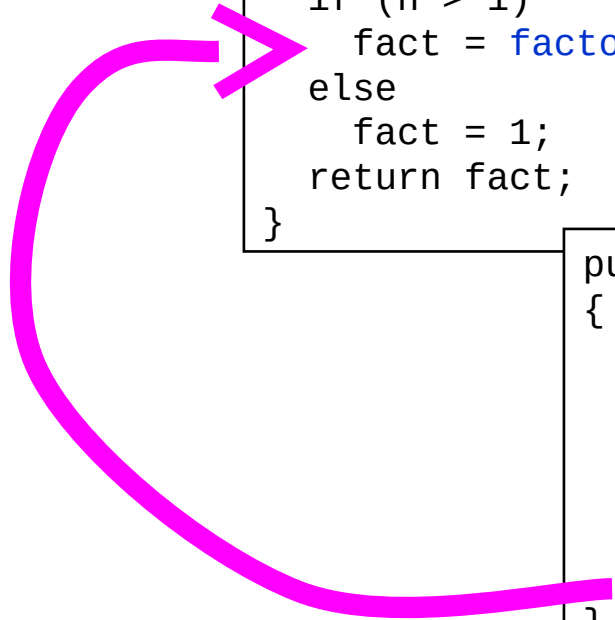
```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

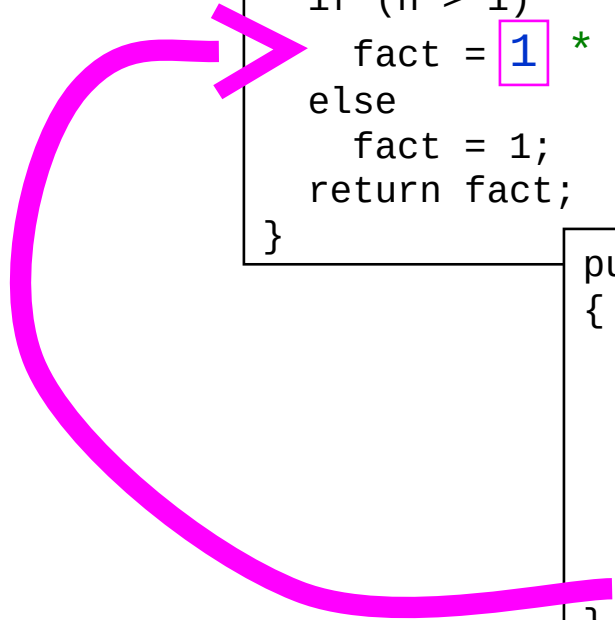
```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```



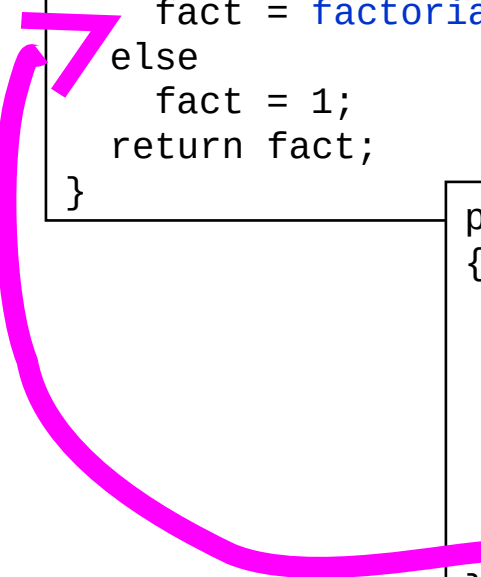
```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```

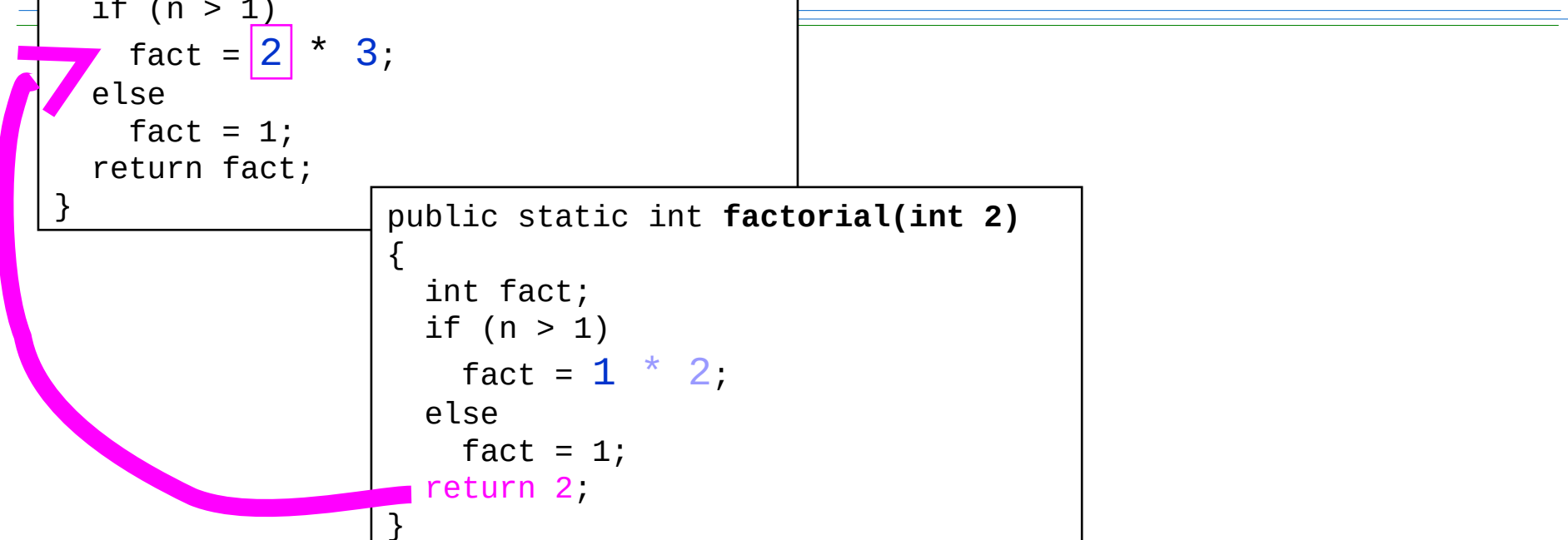


```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```



```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```


```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return fact;
}
```



```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return 6;
}
```

Vuelve al main o a cualquier otro
método que lo haya llamado

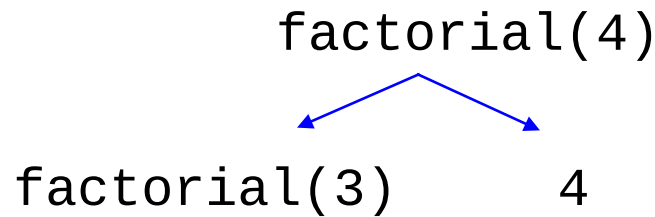


¿Cómo hacer trazas?

- Es una manera de graficar lo que está ejecutándose y saber en qué estado está cada variable que nos interesa, en cada momento
- Hay varias formas de hacer trazas en métodos recursivos
 - **Árbol**
 - **Cajas**
 - **Pila**

Traza de Ejecución (descomposición)

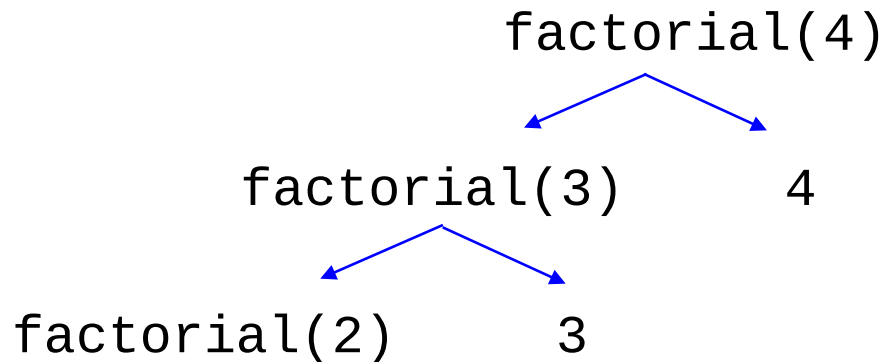
```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de tipo árbol

Traza de Ejecución (descomposición)

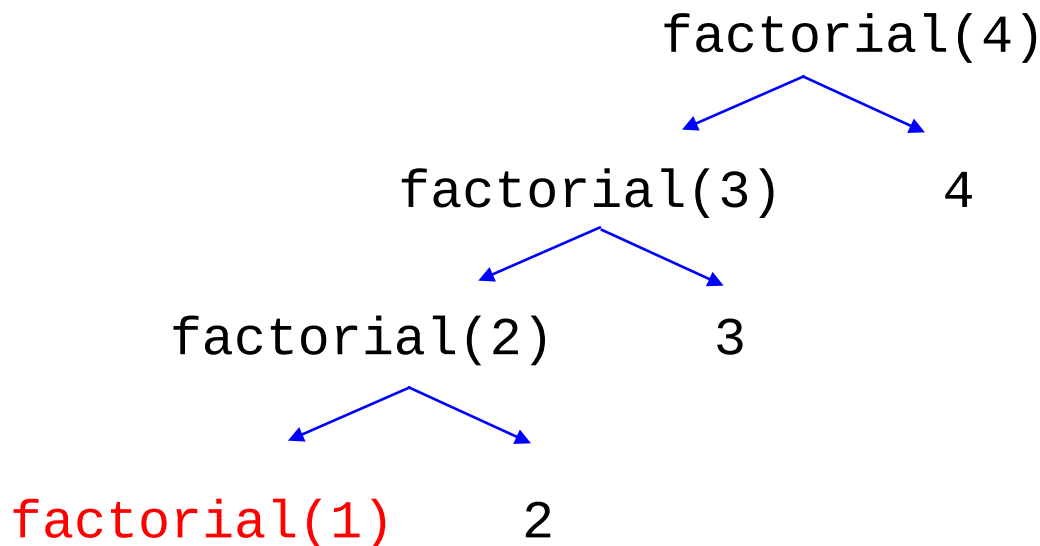
```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de tipo árbol

Traza de Ejecución (descomposición)

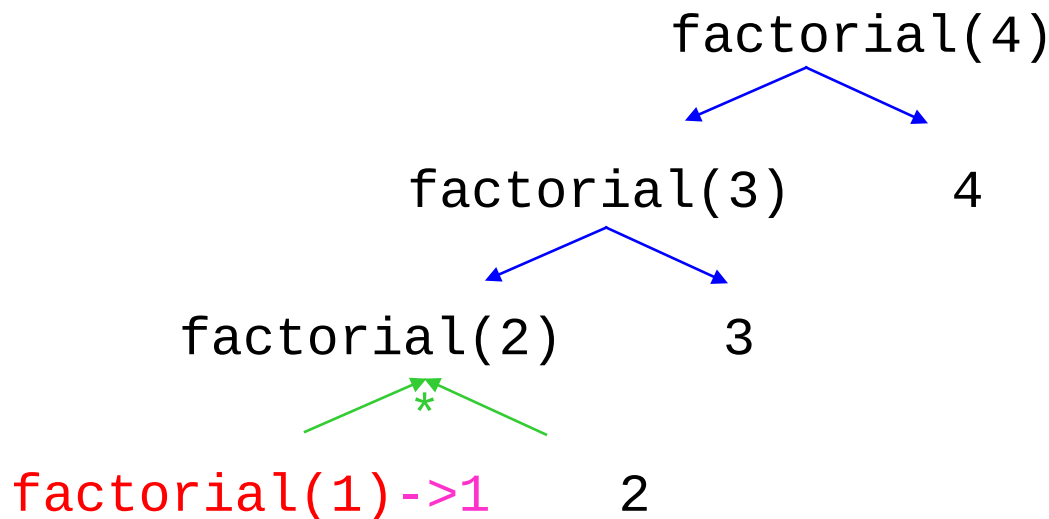
```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de tipo árbol

Traza de Ejecución (composición)

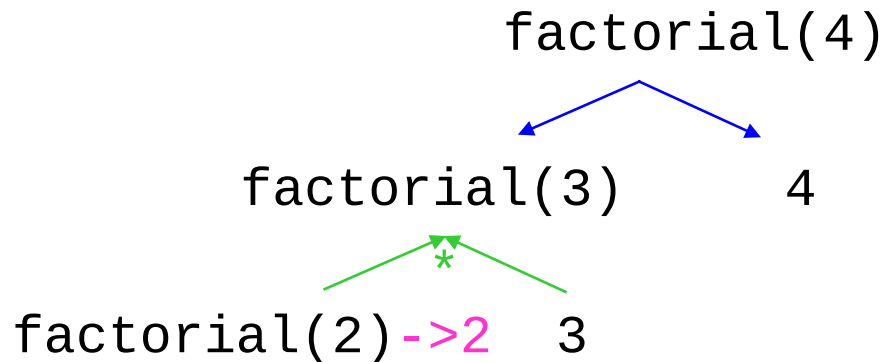
```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de tipo árbol

Traza de Ejecución (composición)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de tipo árbol

Traza de Ejecución (composición)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

factorial(4)
 *
factorial(3) -> 6 4

Traza de tipo árbol

Traza de Ejecución (composición)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

factorial(4) -> 24

Traza de tipo árbol

Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

factorial(4)

n

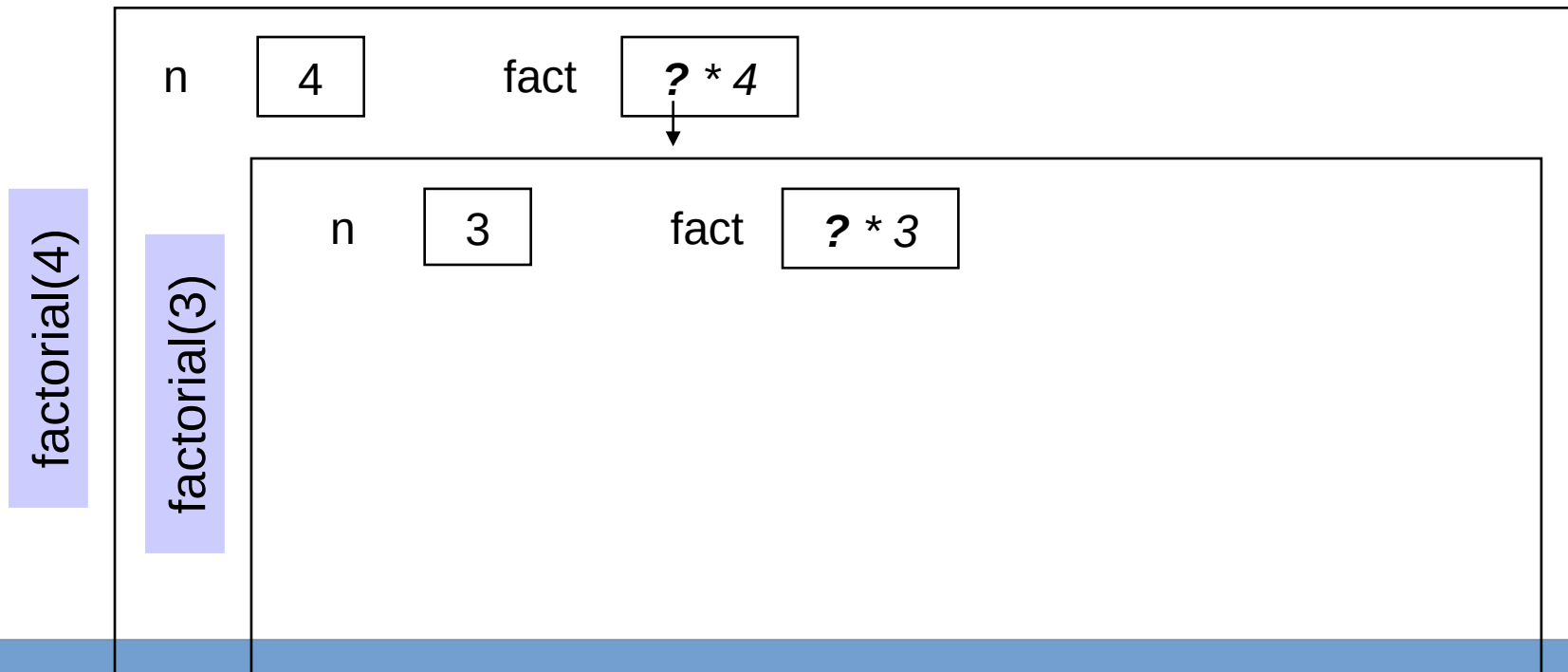
4

fact

? * 4

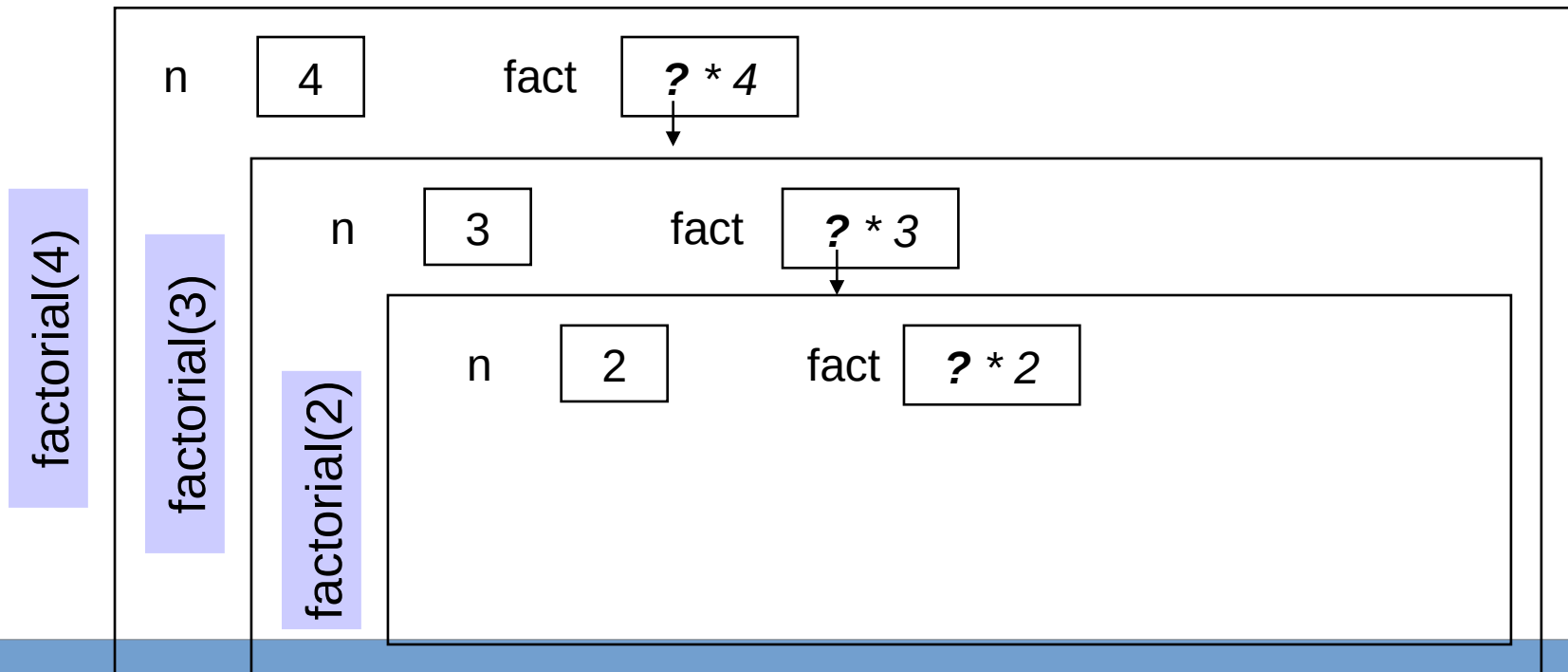
Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



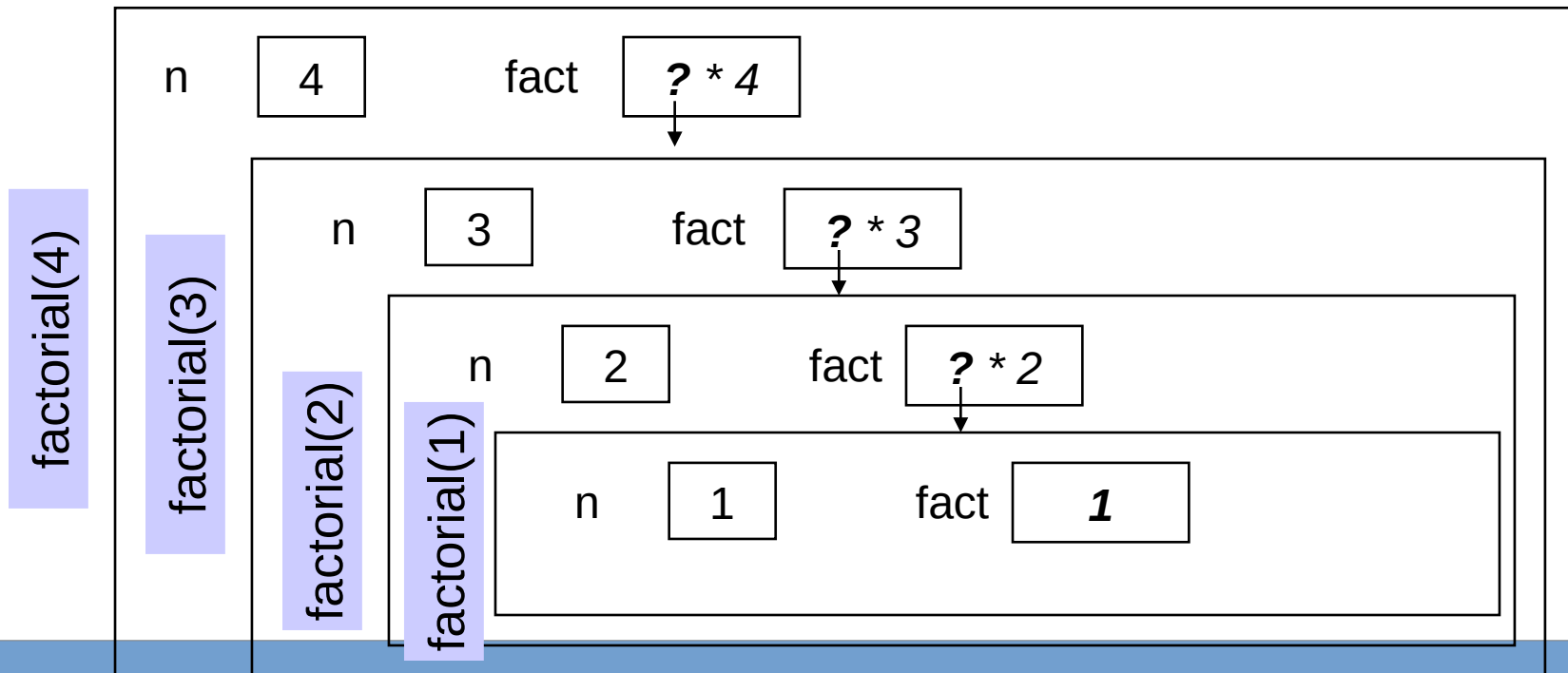
Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



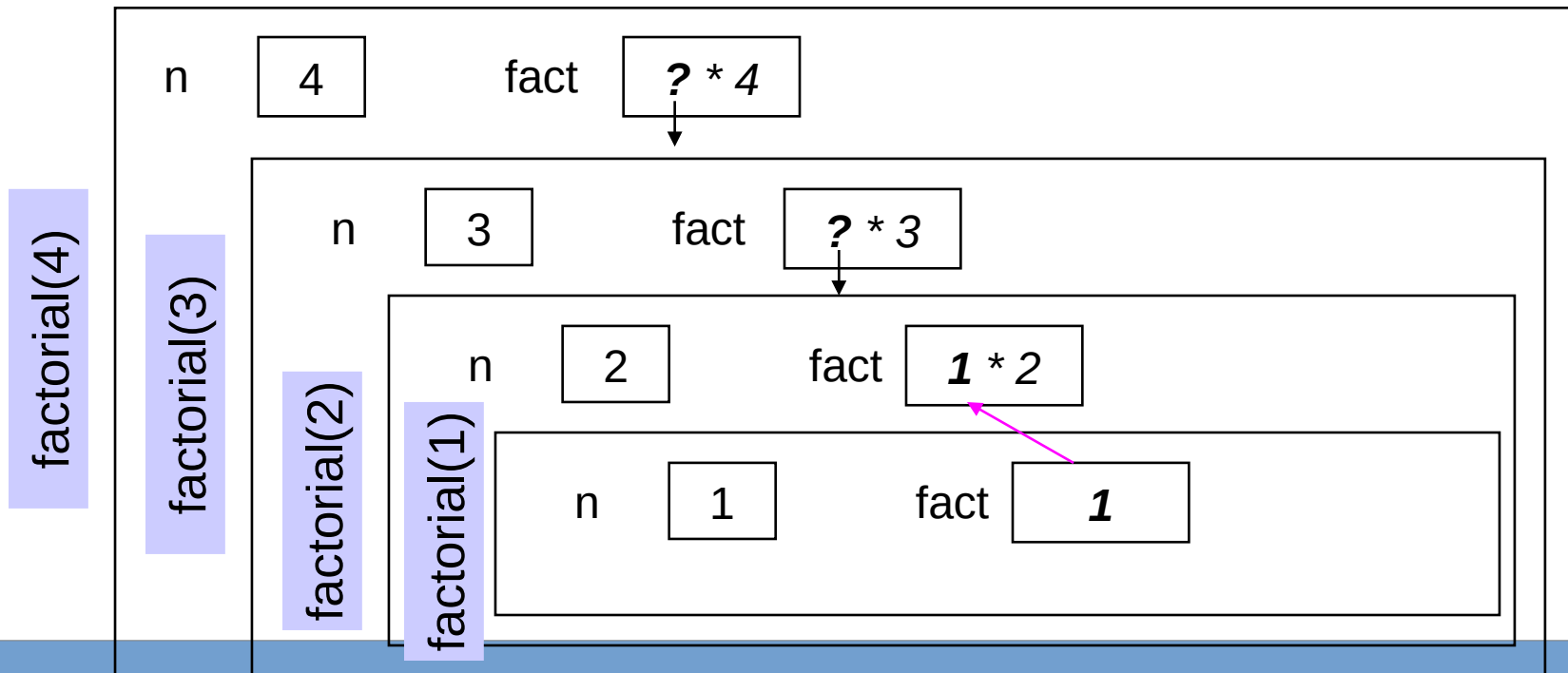
Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



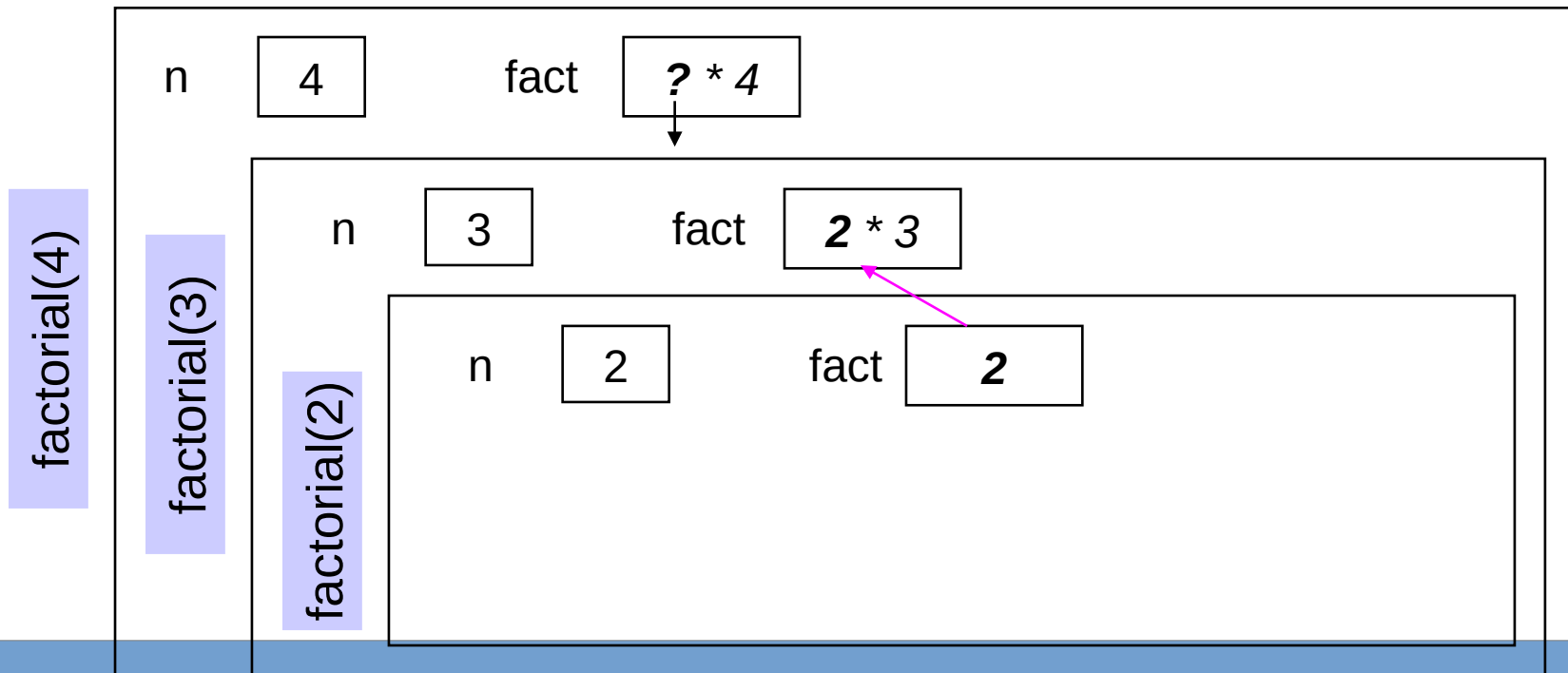
Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



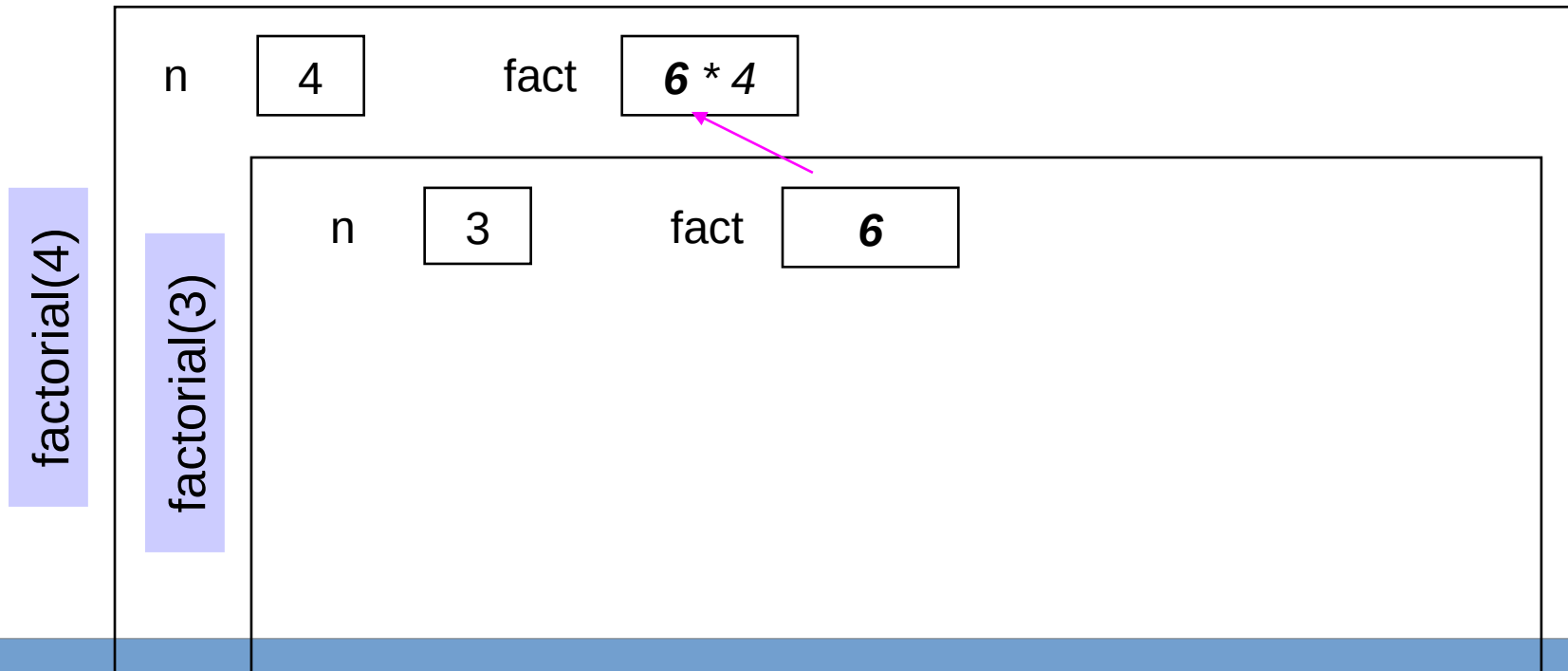
Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```



Traza de Ejecución (con cajas)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

*Se retorna al programa o
método llamador*

n

4

fact

24

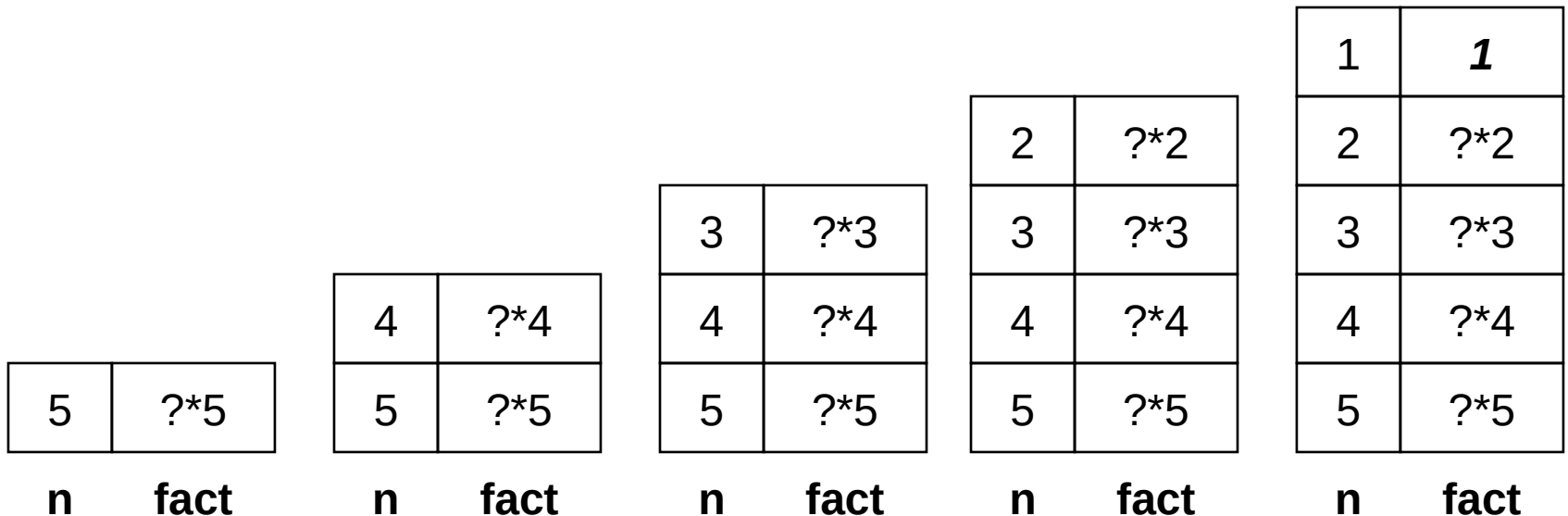
factorial(4)

Traza de Ejecución (con Pila)

factorial(5)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

Llamada recursiva



Traza de Ejecución (con Pila)

factorial(5)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // caso recursivo
        fact = factorial(n - 1) * n;
    else // caso base
        fact = 1;
    return fact;
}
```

Llamada recursiva

1	1
2	1*2
3	?*3
4	?*4
5	?*5

n fact

2	2
3	2*3
4	?*4
5	?*5

n fact

3	6
4	6*4
5	?*5

n fact

4	24
5	24*5

n fact

5	120
---	------------

n fact

Hacer

- Programar un algoritmo recursivo que permita sumar los elementos de un arreglo.



- Programar un algoritmo recursivo que permita invertir un número. **Ejemplo:** Entrada: 123 Salida: 321

$$F(123) = (3 * 100 + F(12))$$

$$(123 \% 10) - (123 / 10)$$

$$F(12) = (2 * 10) + F(1)$$

$$F(1) = 1$$

- caso base

Hacer

- Programar un algoritmo recursivo que permita sumar los elementos de un arreglo.

Cantidad de elementos: n



¿Cual es el caso base?

$n = 0$

SI ($n = 0$) ENTONCES

.....

FIN SI

$n = \text{longitud del arreglo}$

SI ($n = \text{longitud}(\text{arr}[])$) ENTONCES

.....

FIN SI

De acuerdo al caso base, corta en $n = 0$ o cuando $n = \text{longitud del arreglo}$

Hacer

- Algoritmo recursivo que suma los elementos de un arreglo.

```
public static int suma_arreglo(int arr[], int n) {  
    int sum;  
    if (n == 0)  
        sum = arr[n];  
    else  
        sum = suma_arreglo(arr, n-1) + arr[n];  
    return sum  
}
```

MODULO suma_arreglo(arr[], int n)
(*arr es arreglo de enteros, n es un entero retorna un entero*)

```
entero sum  
    SI (n = 0) ENTONCES  
        sum ← arr [n];  
    SINO  
        sum ← suma_arreglo(arr, n-1) + arr[n];  
    FIN SI  
    SALE sum  
FIN MODULO
```

5	2	1	4	3	0	6	7
---	---	---	---	---	---	---	---

Ejemplo: Números Fibonacci

- El n-ésimo número Fibonacci es la suma de los dos números Fibonacci inmediatamente anteriores
 - 0, 1, 1, 2, 3, 5, 8, 13, ...
- Diseño Recursivo:
 - Descomposición & Composición
 - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
 - Caso Base:
 - $\text{fibonacci}(1) = 0$
 - $\text{fibonacci}(2) = 1$

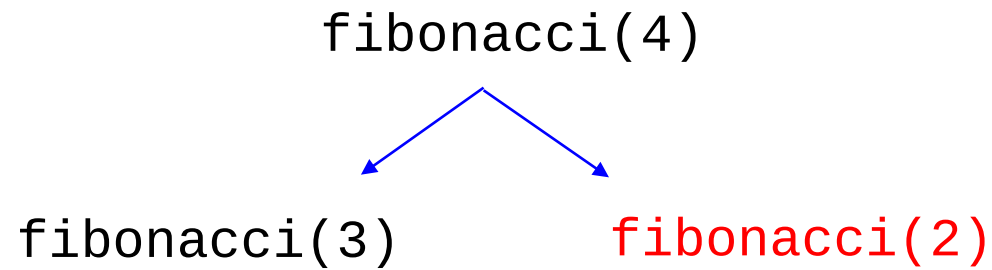
Método fibonacci

```
public static int fibonacci(int n)
{
    int fib;
    if (n > 2)
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}
```

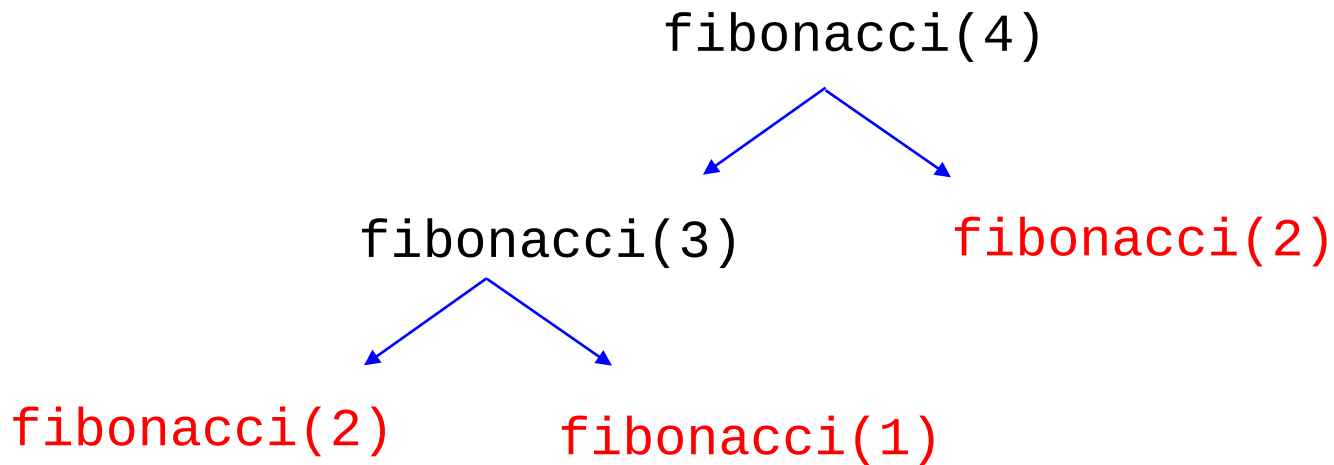
The diagram illustrates the recursive Fibonacci method with four callouts:

- CASO RECURSIVO** (yellow box) points to the `if (n > 2)` condition.
- composición** (green box) points to the recursive call `fibonacci(n-1) + fibonacci(n-2)`.
- descomposición** (blue box) points to the recursive call `fibonacci(n-2)`.
- CASO BASE** (orange box) points to the `else if (n == 2)` and `else` branches, which represent the base cases of the recursion.

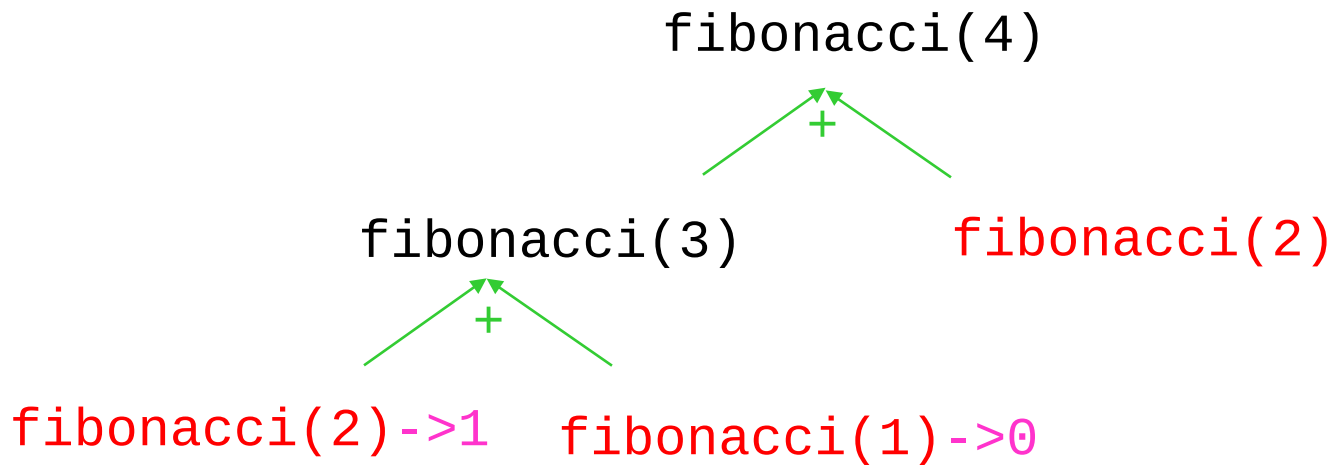
Traza de Ejecución (descomposición)



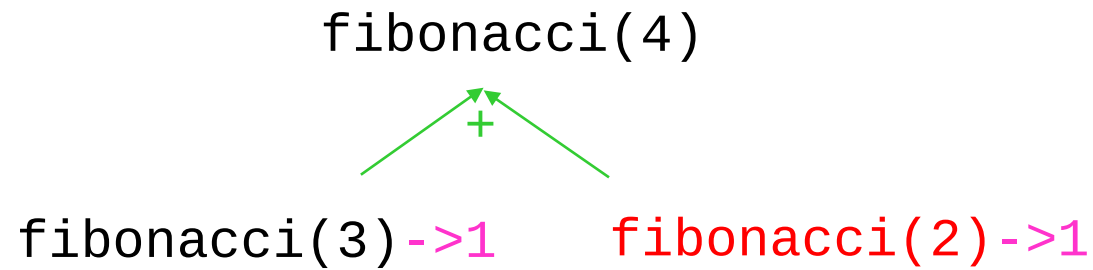
Traza de Ejecución (descomposición)



Traza de Ejecución (composición)



Traza de Ejecución (composición)



Traza de Ejecución (**composición**)

`fibonacci(4) ->2`

Traza de Ejecución (con cajas)

fibonacci(5)

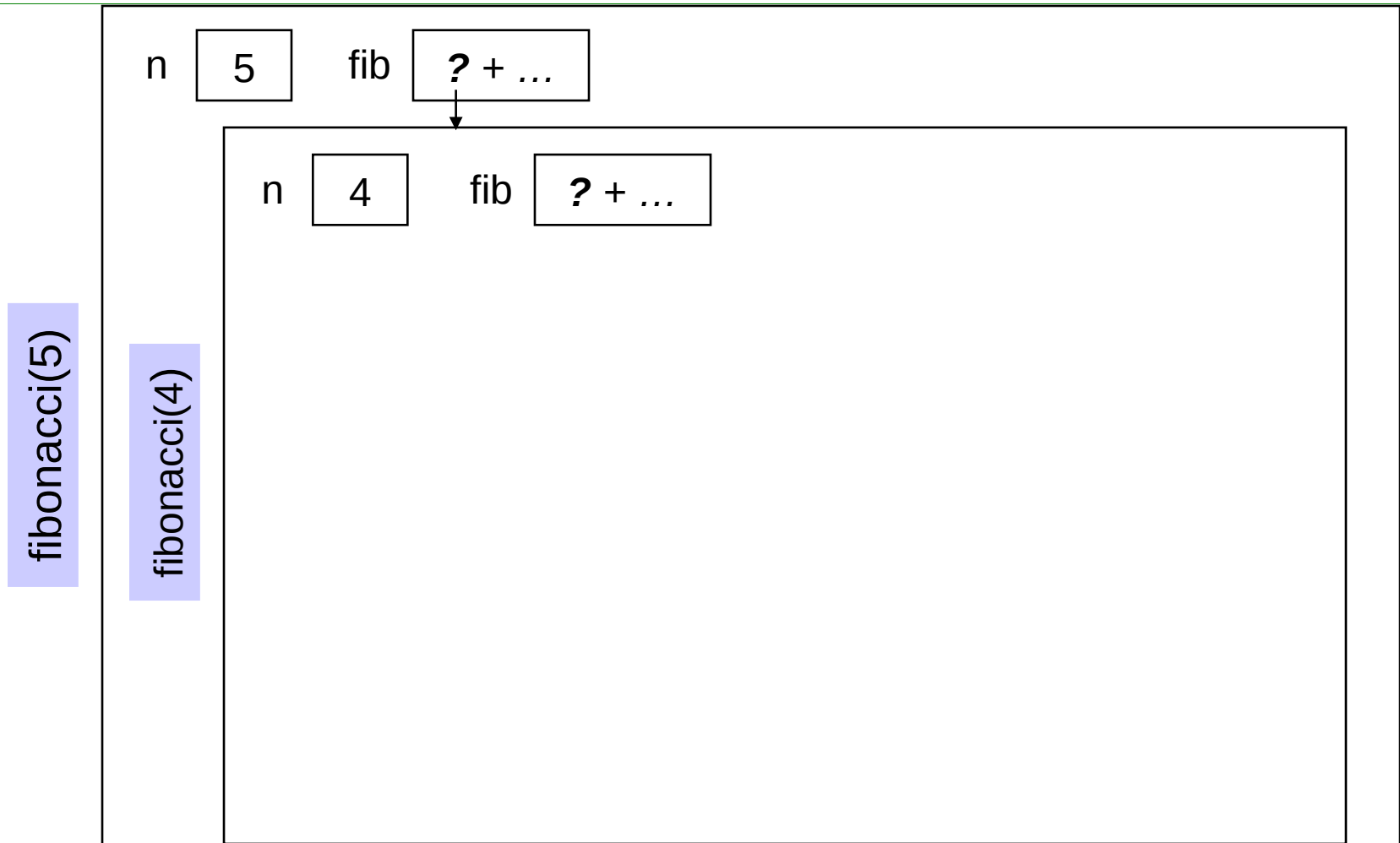
n

5

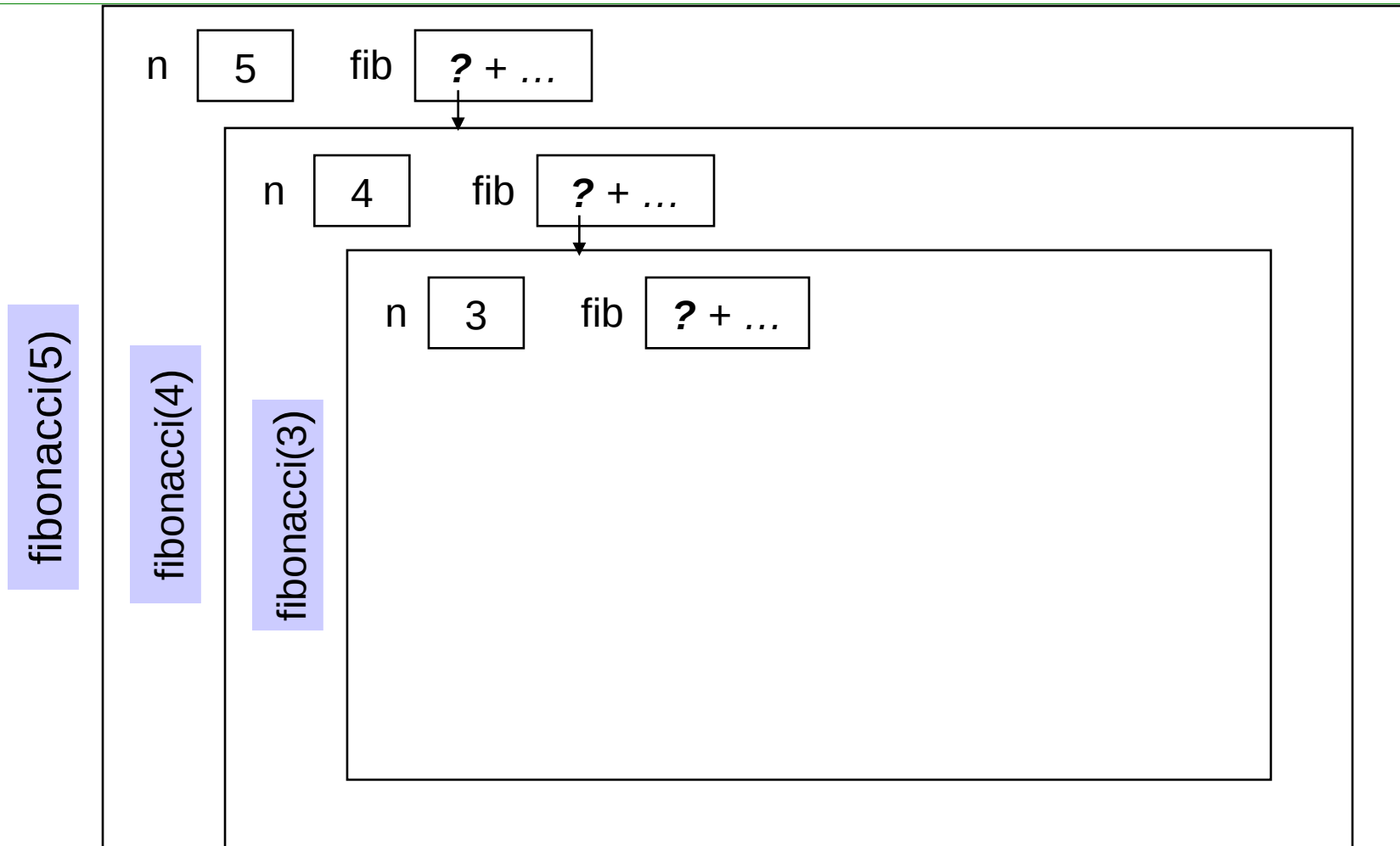
fib

? + ...

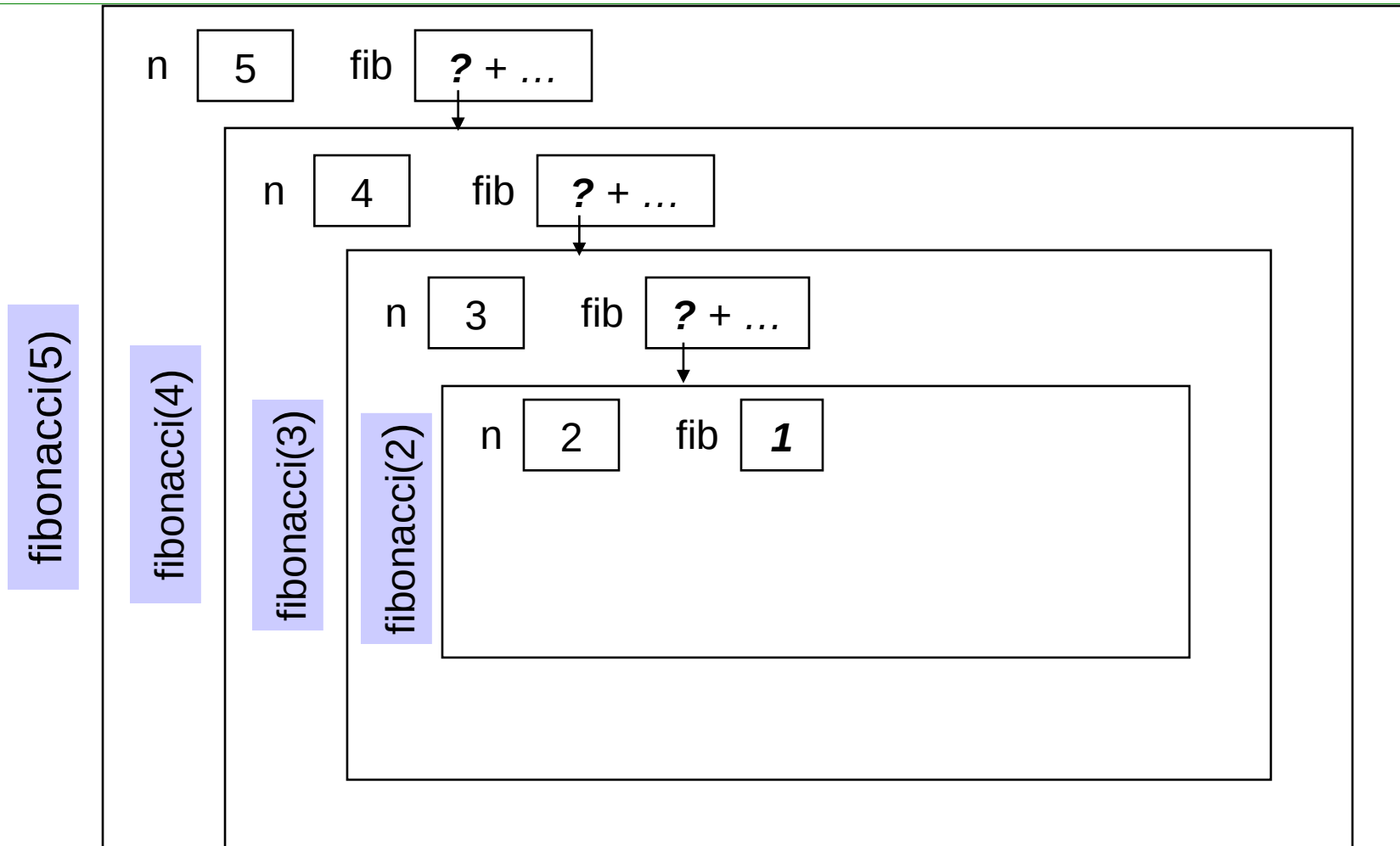
Traza de Ejecución (con cajas)



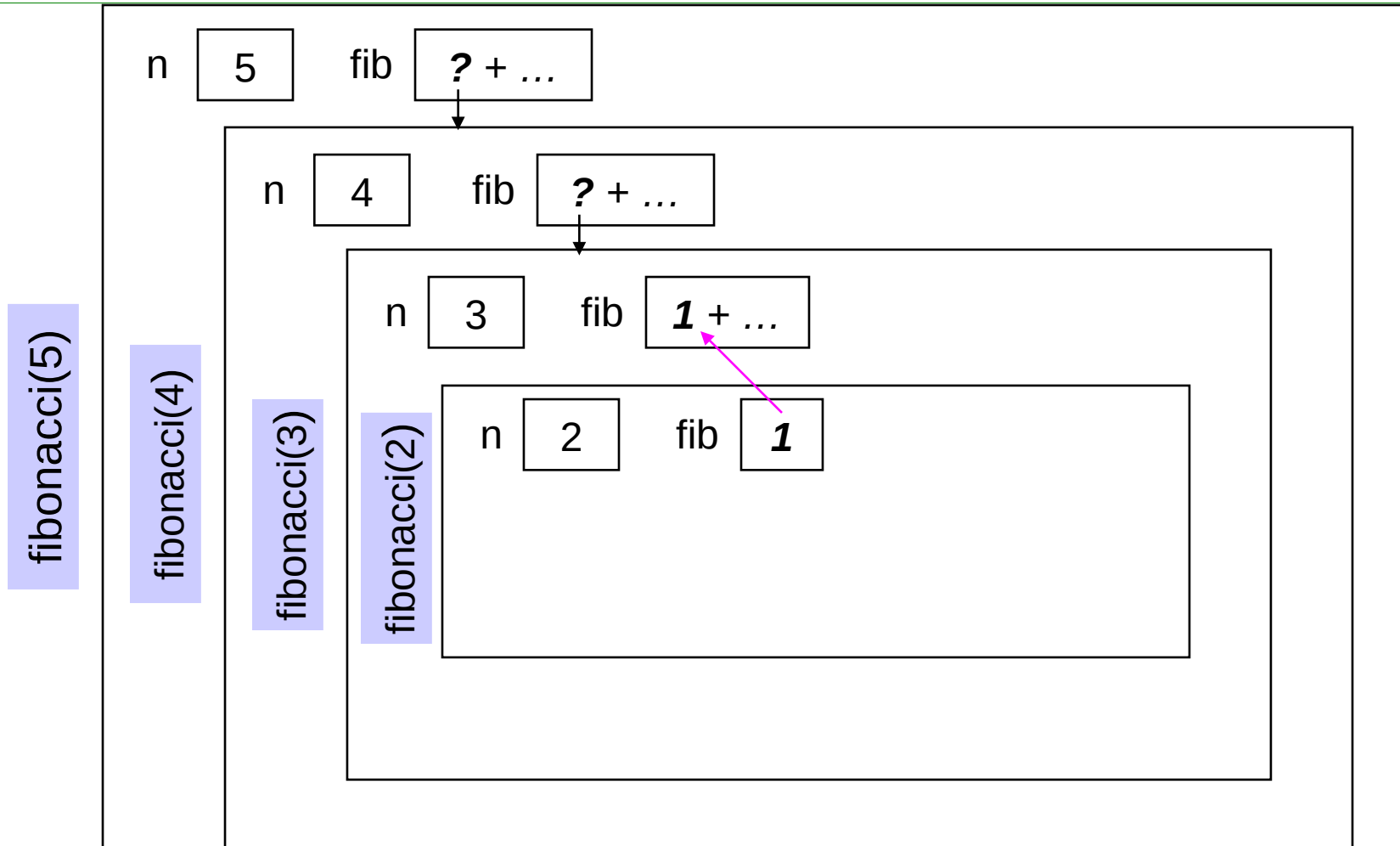
Traza de Ejecución (con cajas)



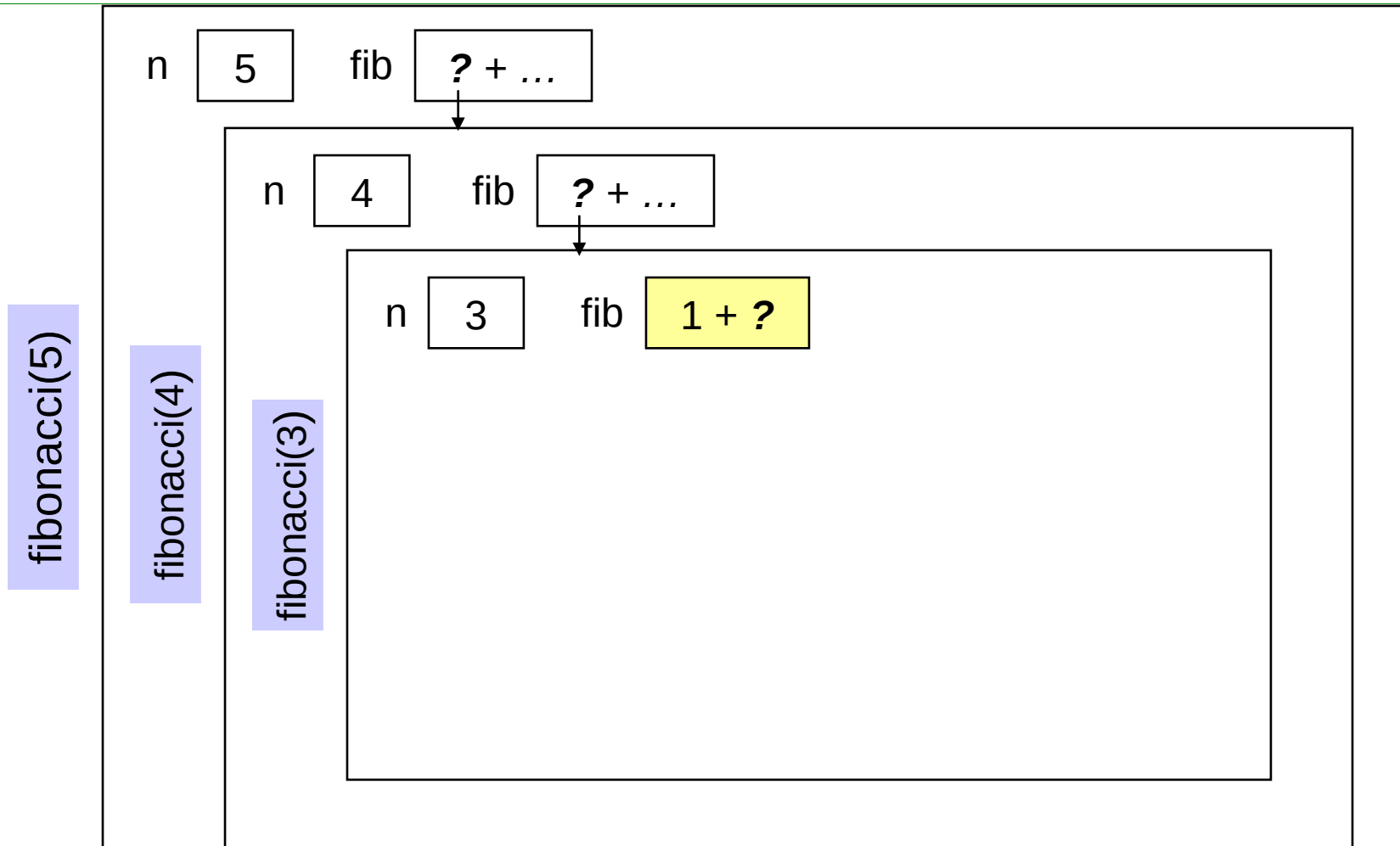
Traza de Ejecución (con cajas)



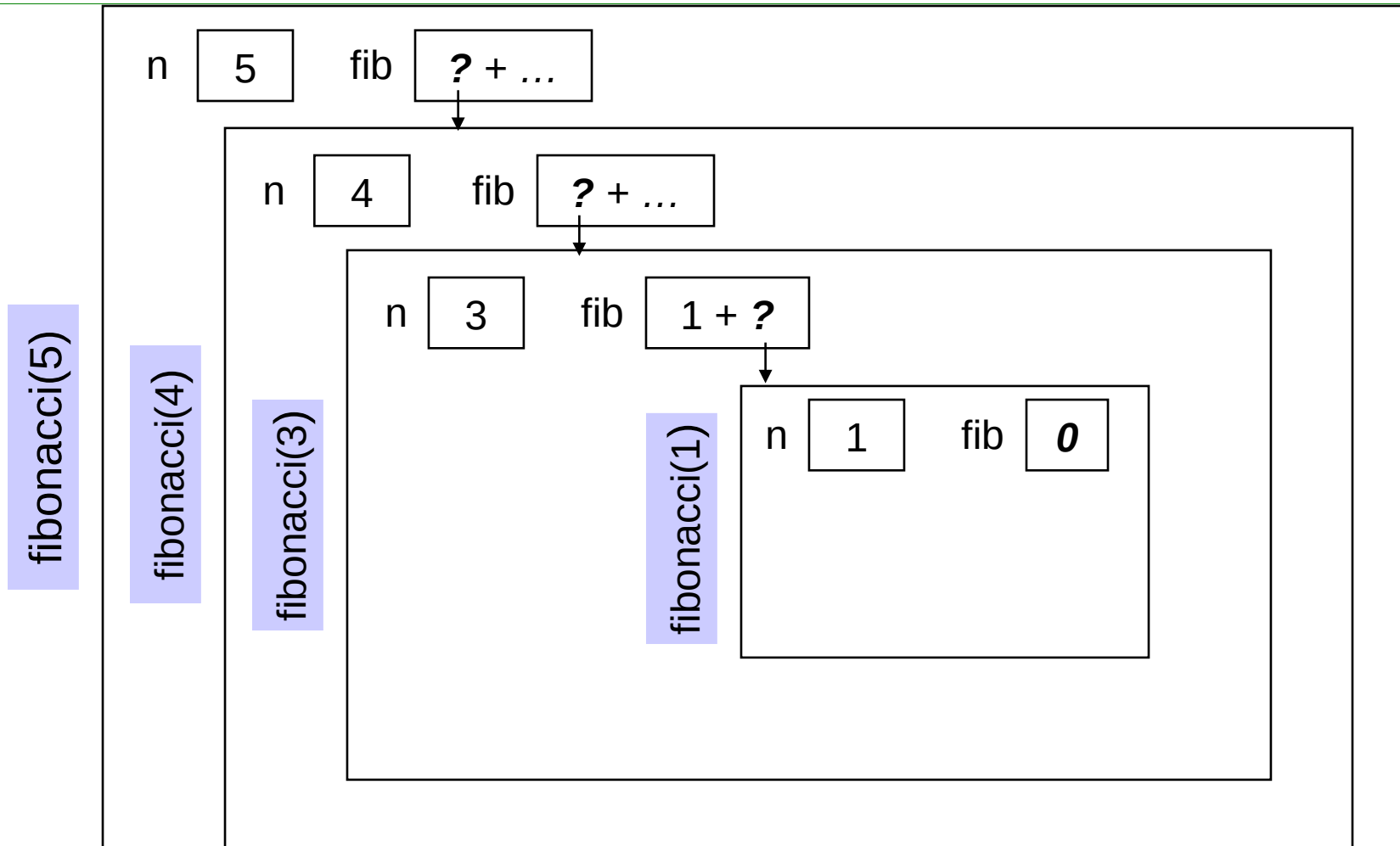
Traza de Ejecución (con cajas)



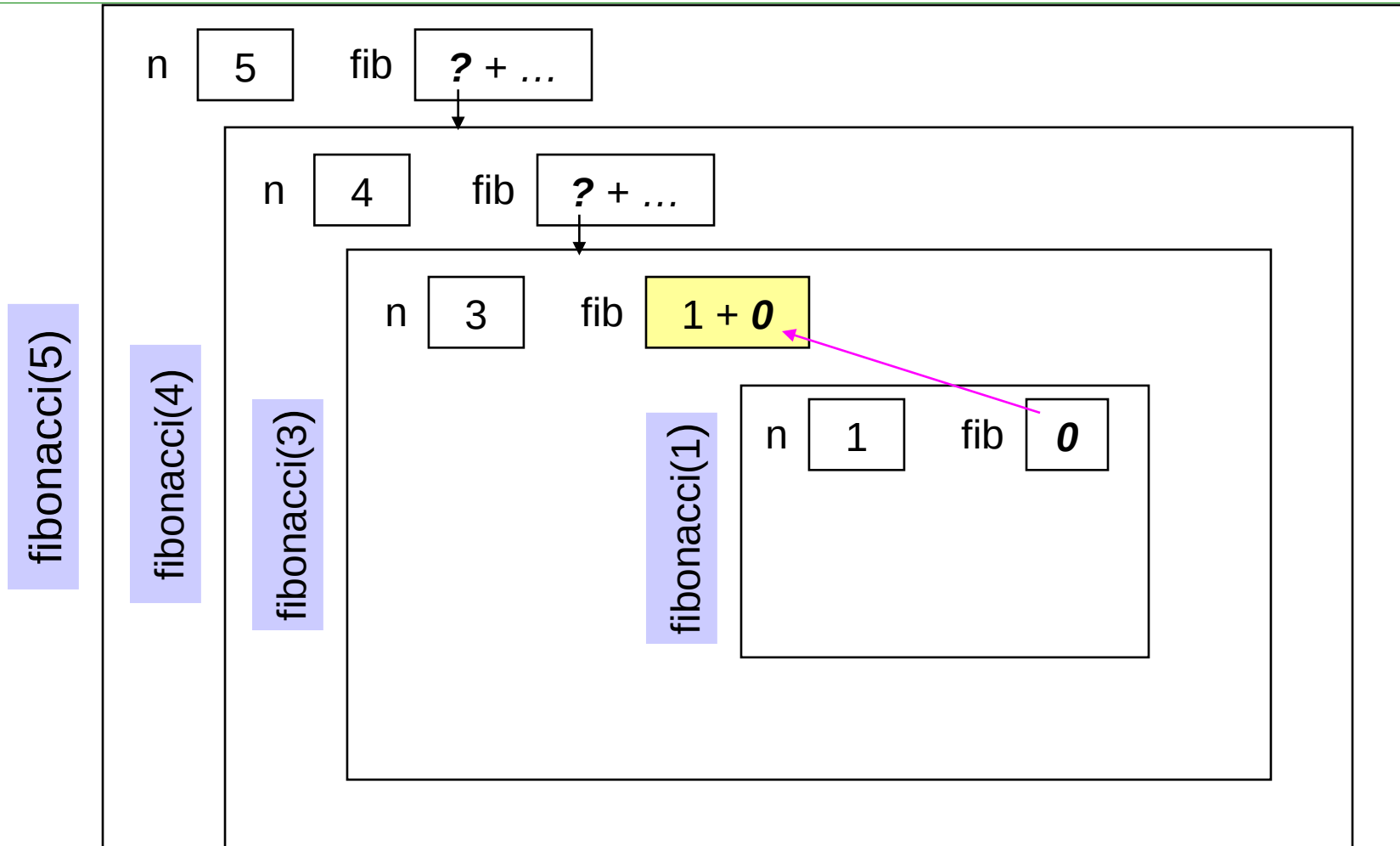
Traza de Ejecución (con cajas)



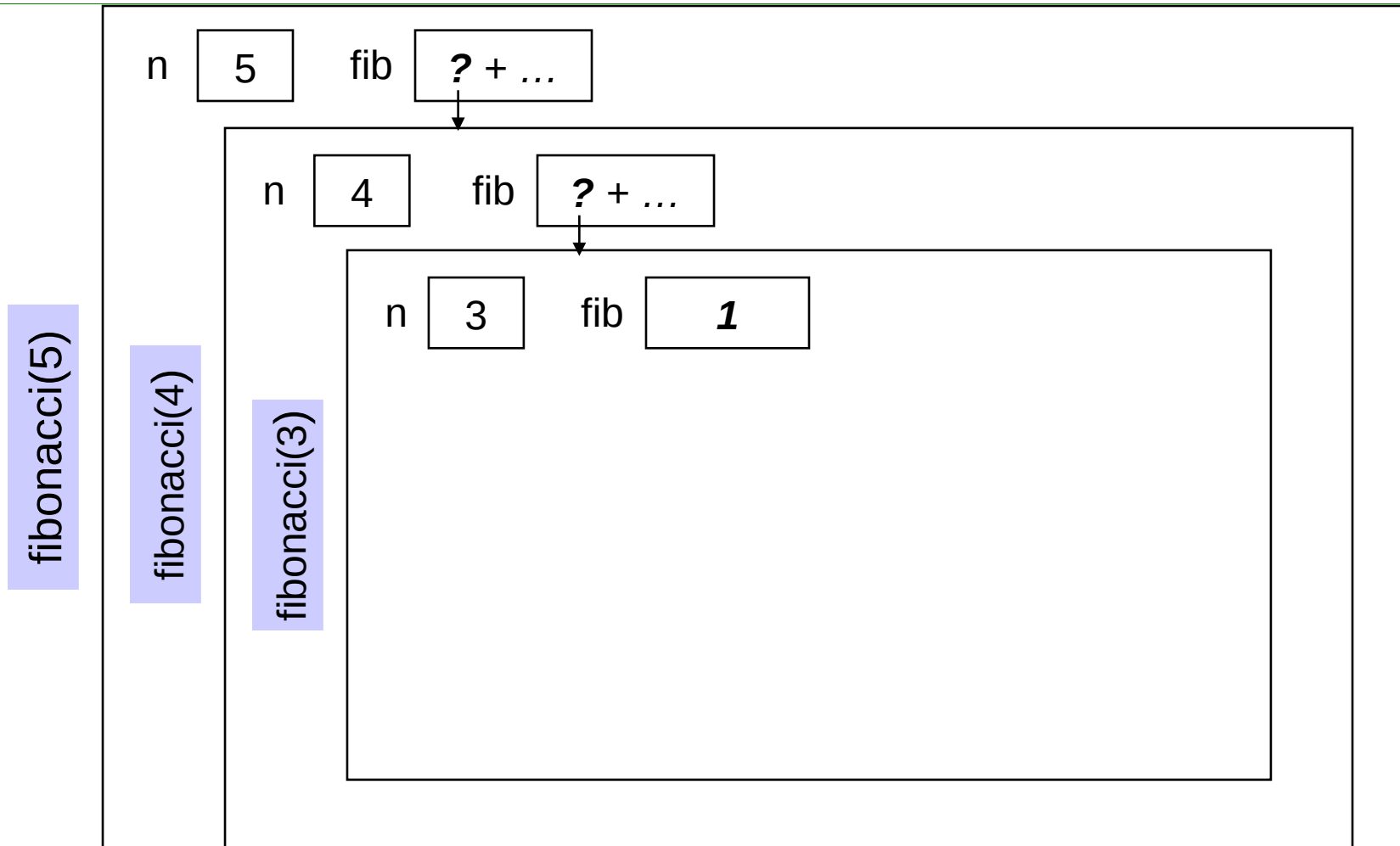
Traza de Ejecución (con cajas)



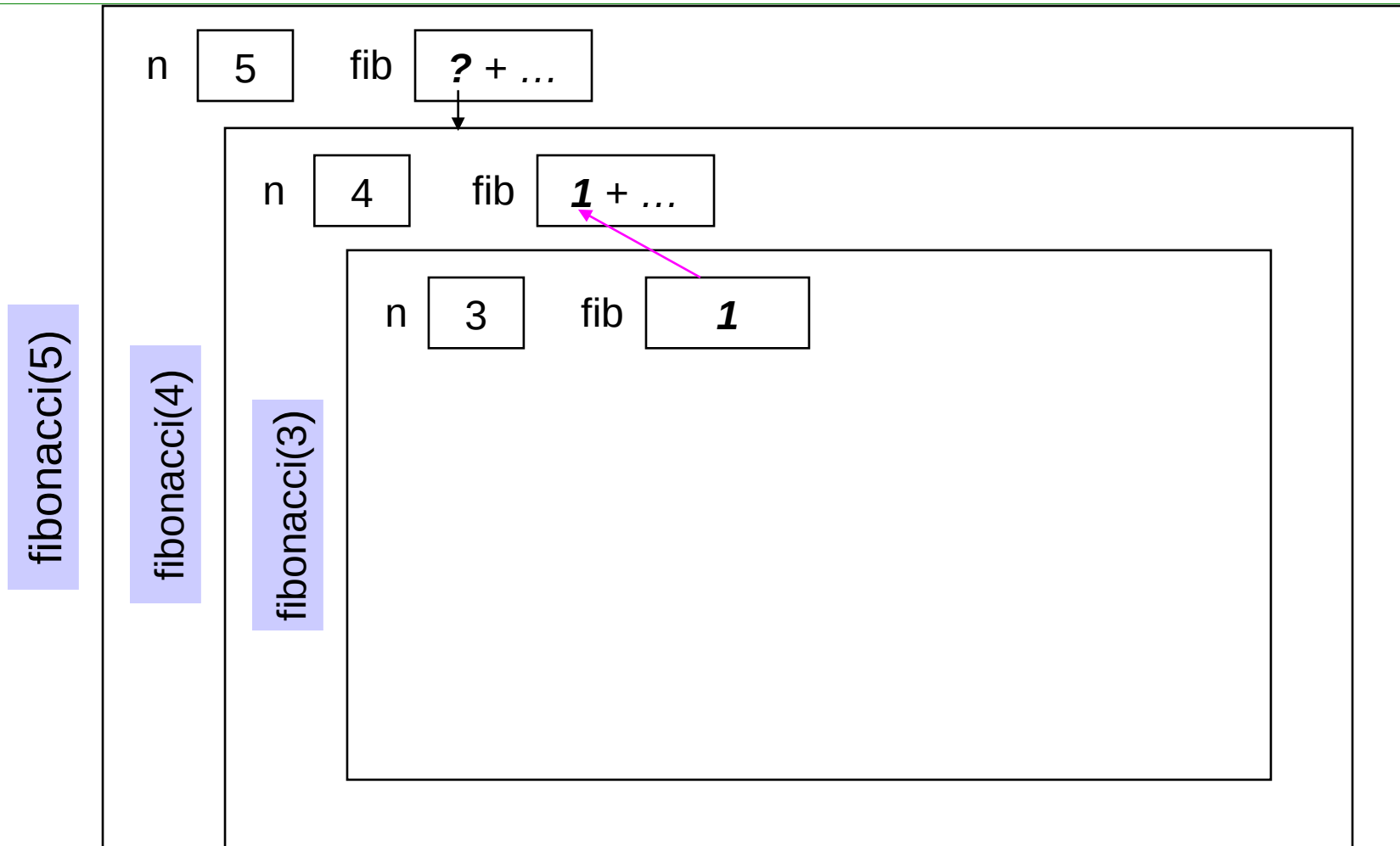
Traza de Ejecución (con cajas)



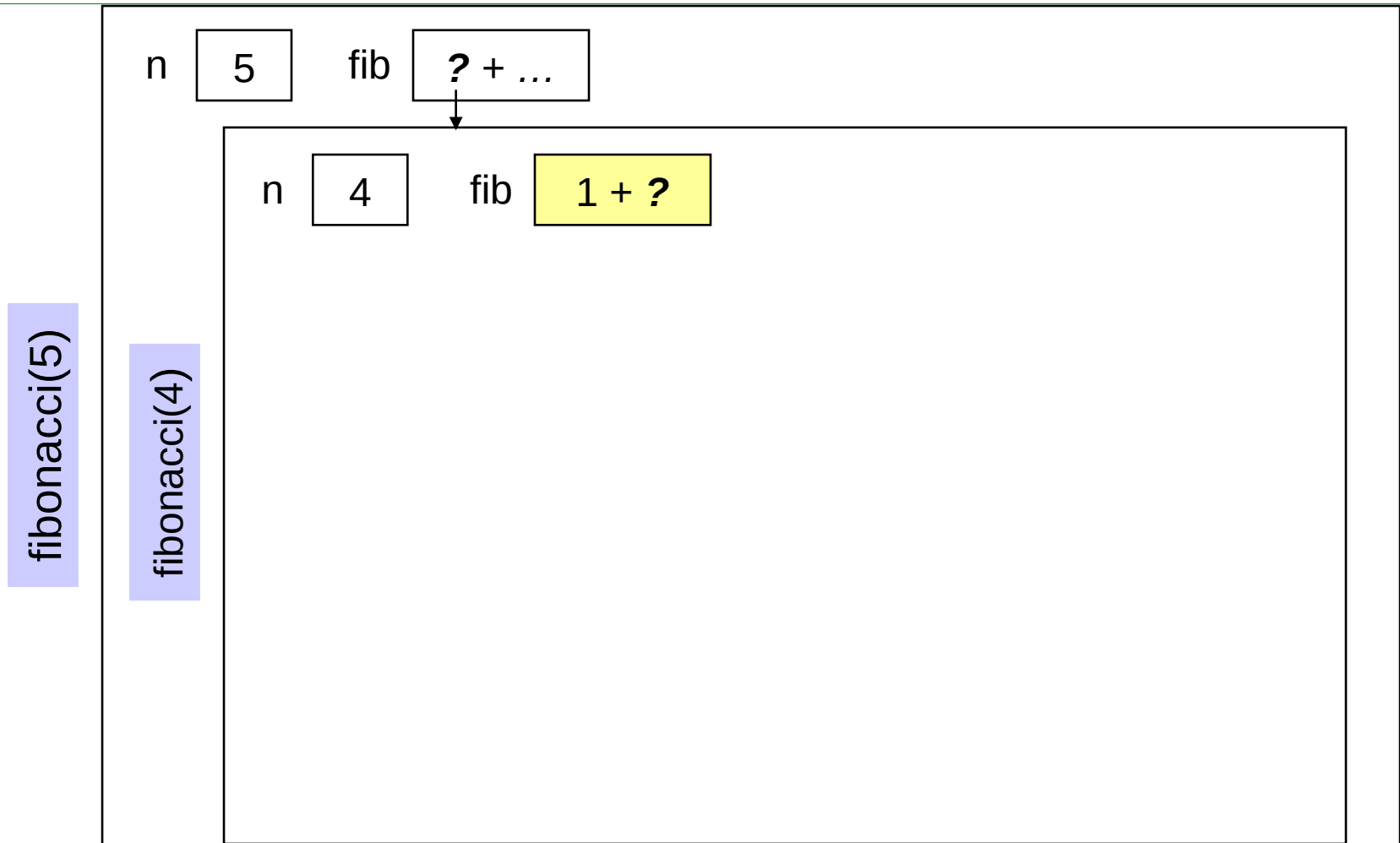
Traza de Ejecución (con cajas)



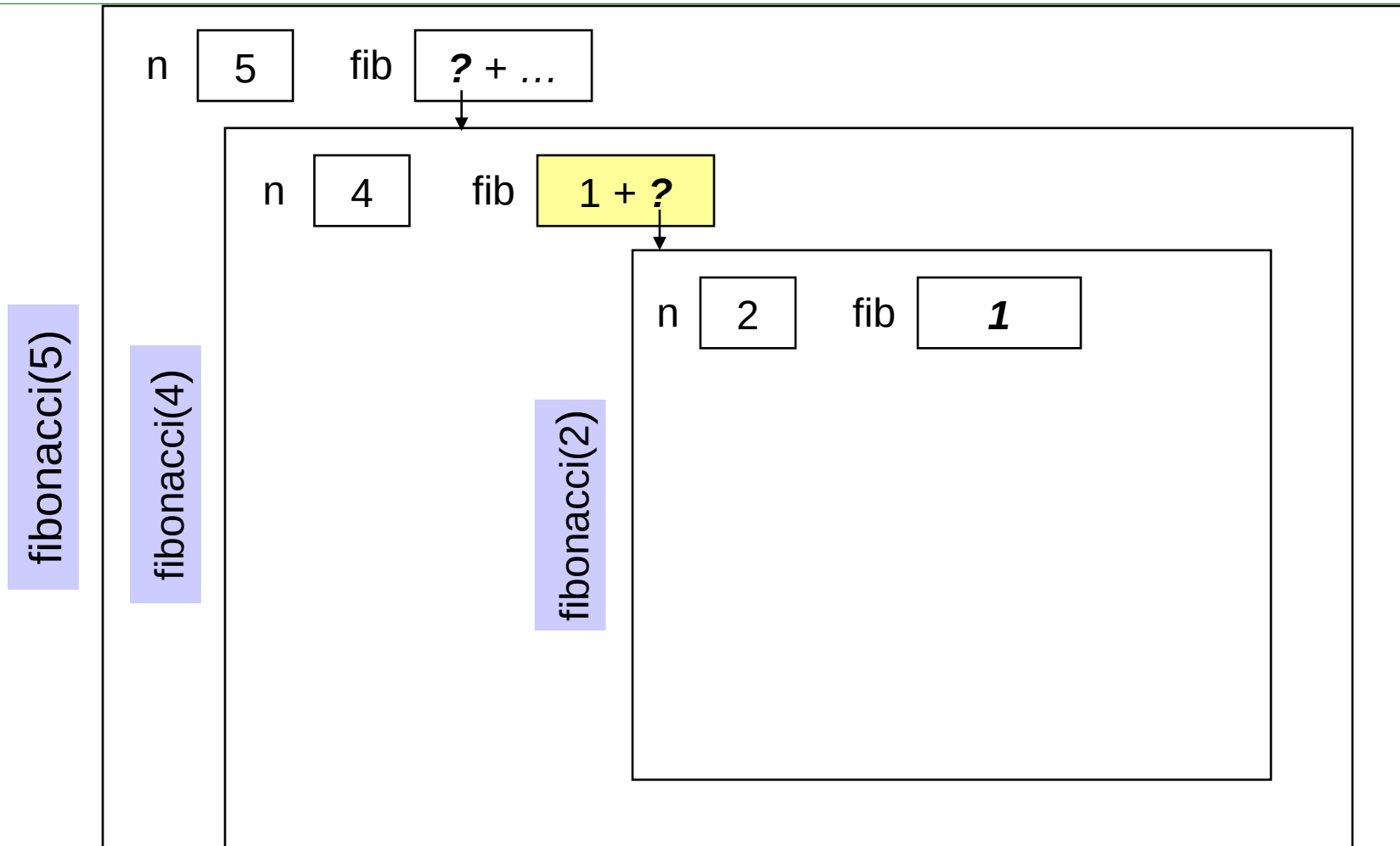
Traza de Ejecución (con cajas)



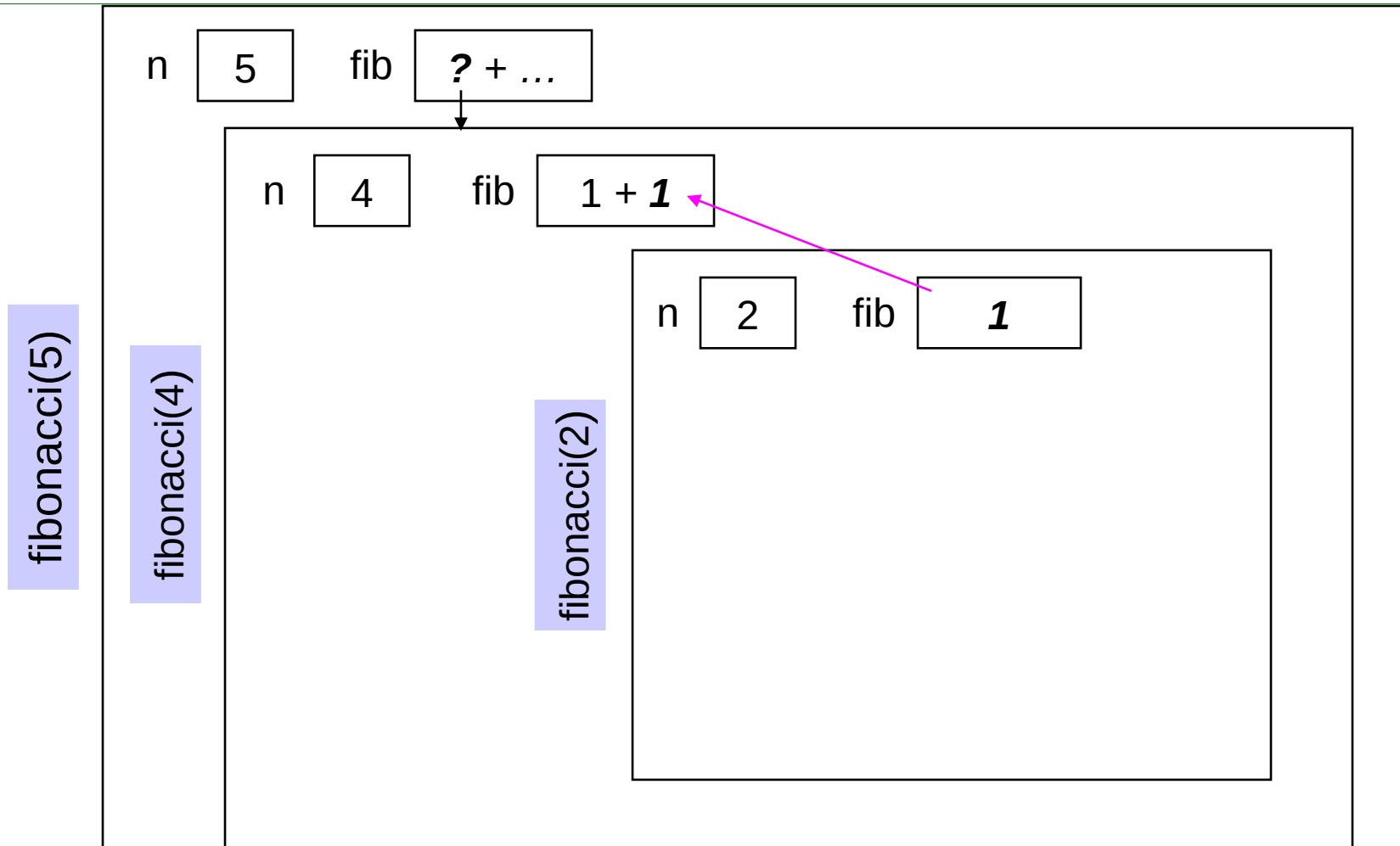
Traza de Ejecución (con cajas)



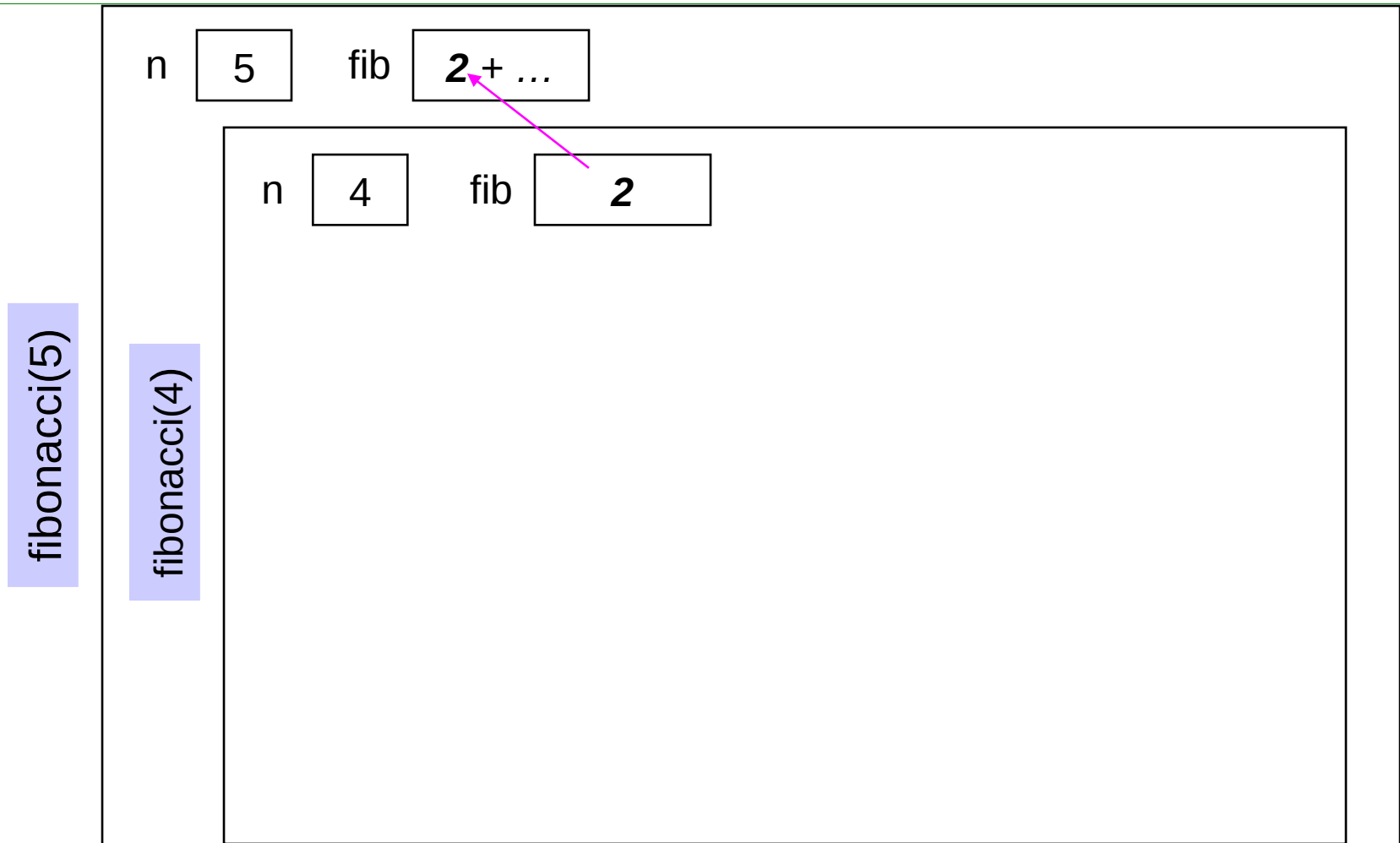
Traza de Ejecución (con cajas)



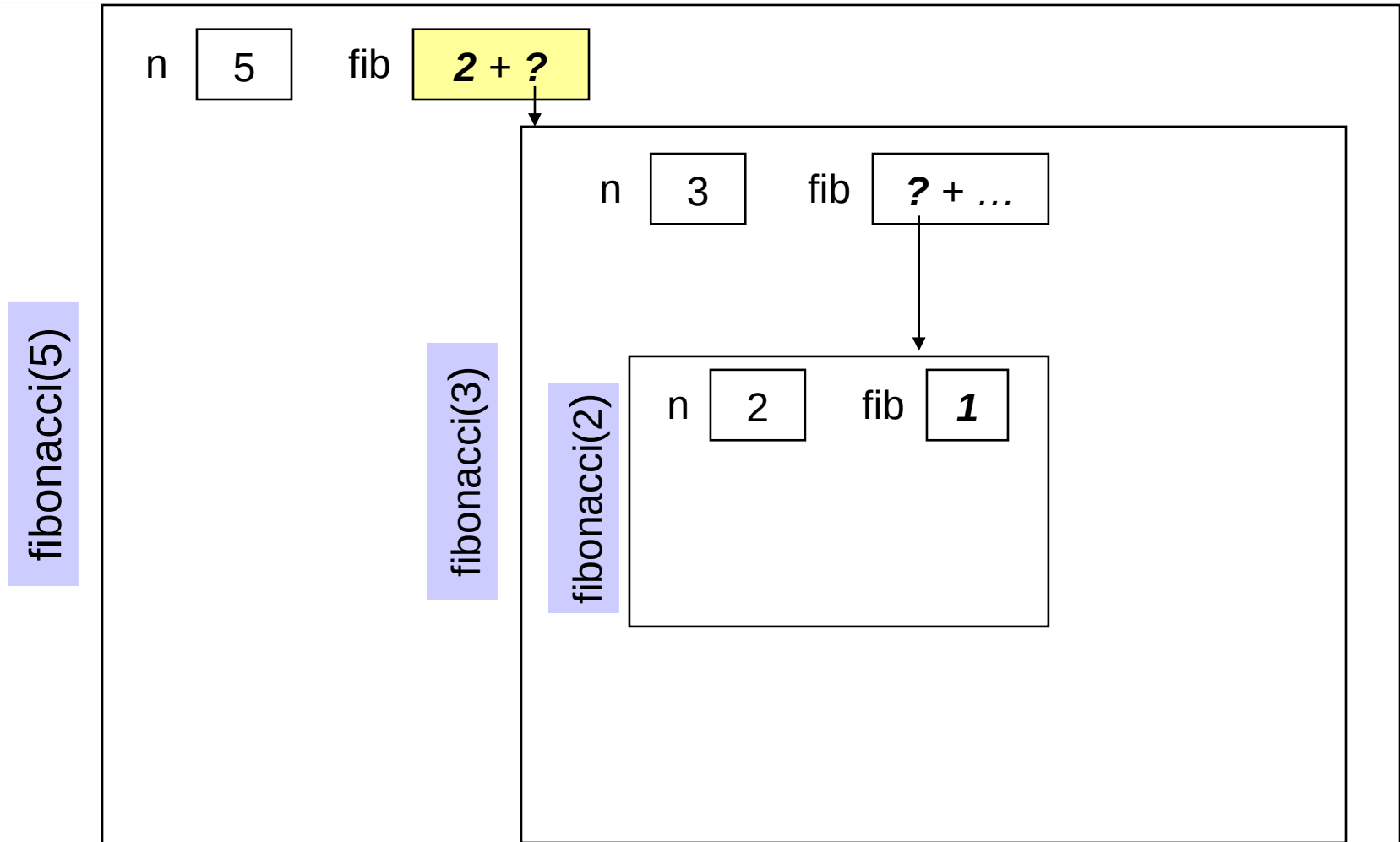
Traza de Ejecución (con cajas)



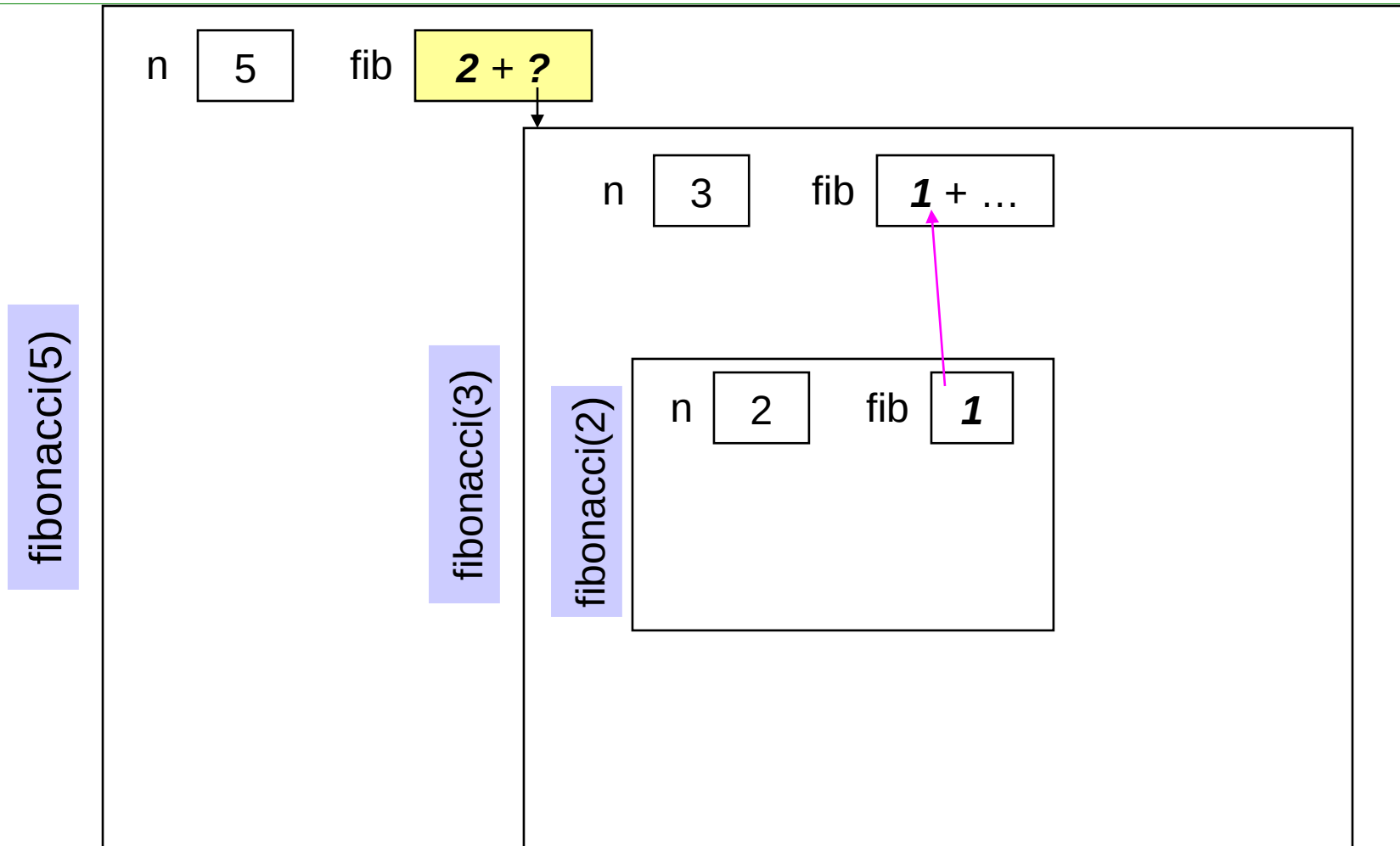
Traza de Ejecución (con cajas)



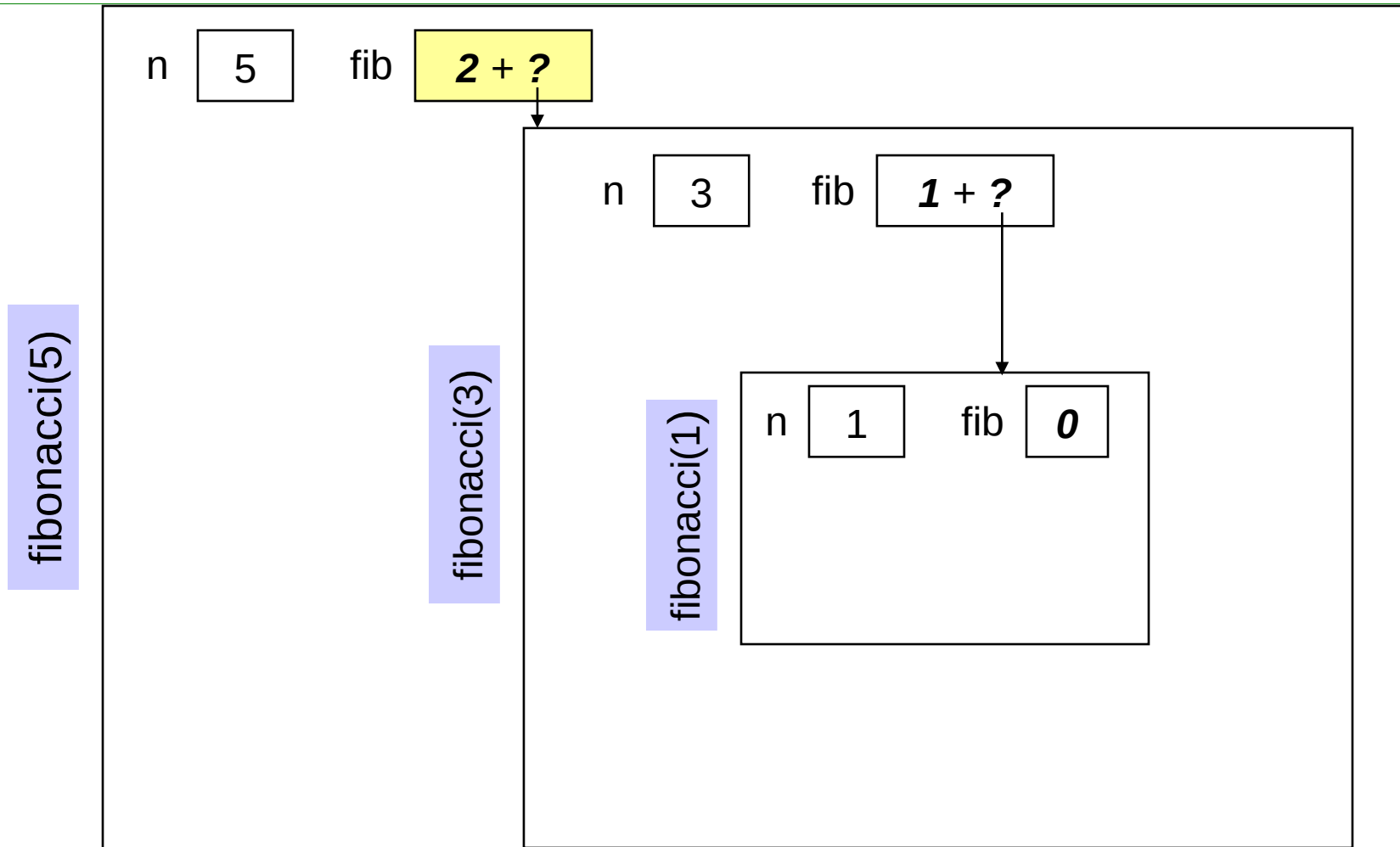
Traza de Ejecución (con cajas)



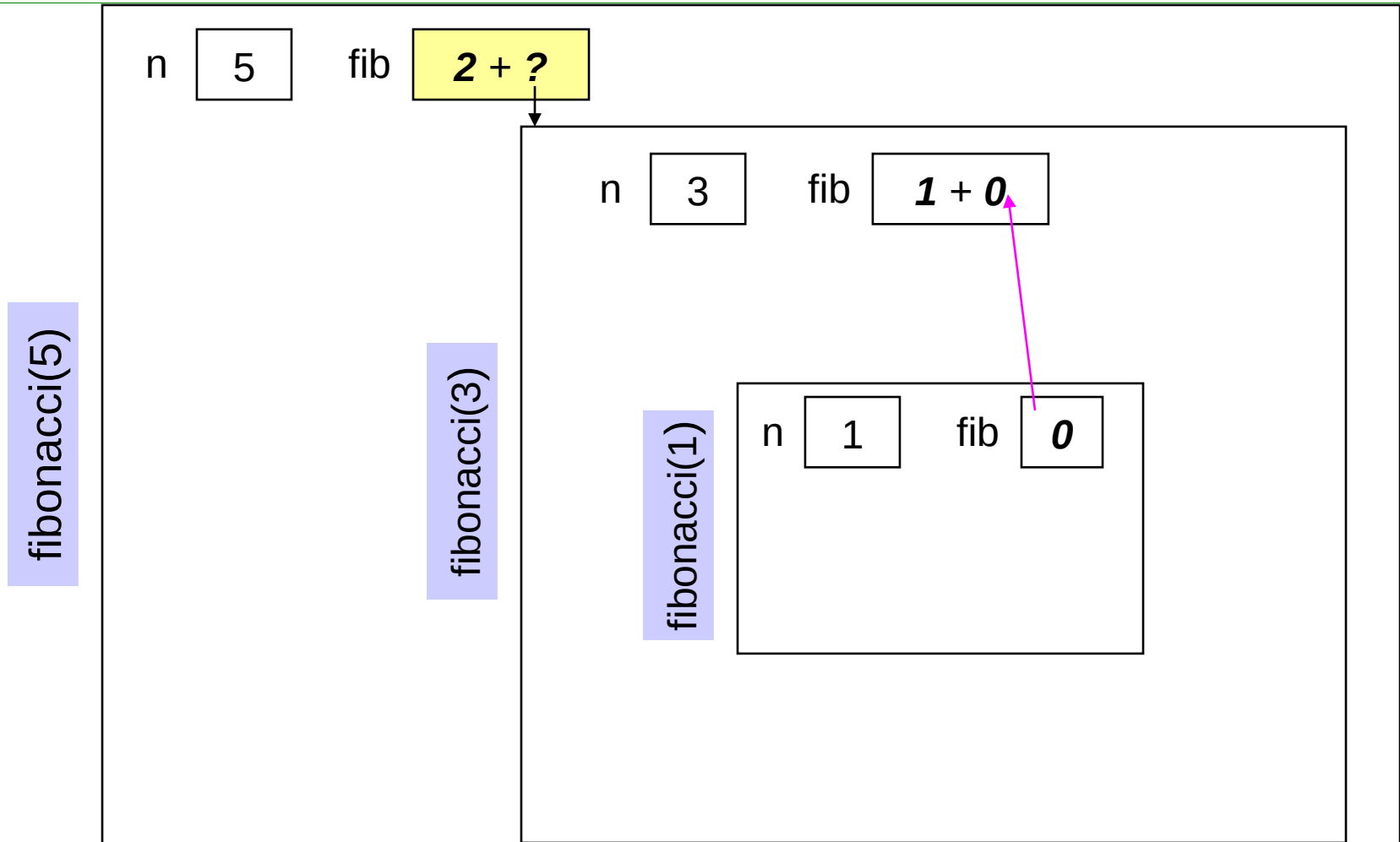
Traza de Ejecución (con cajas)



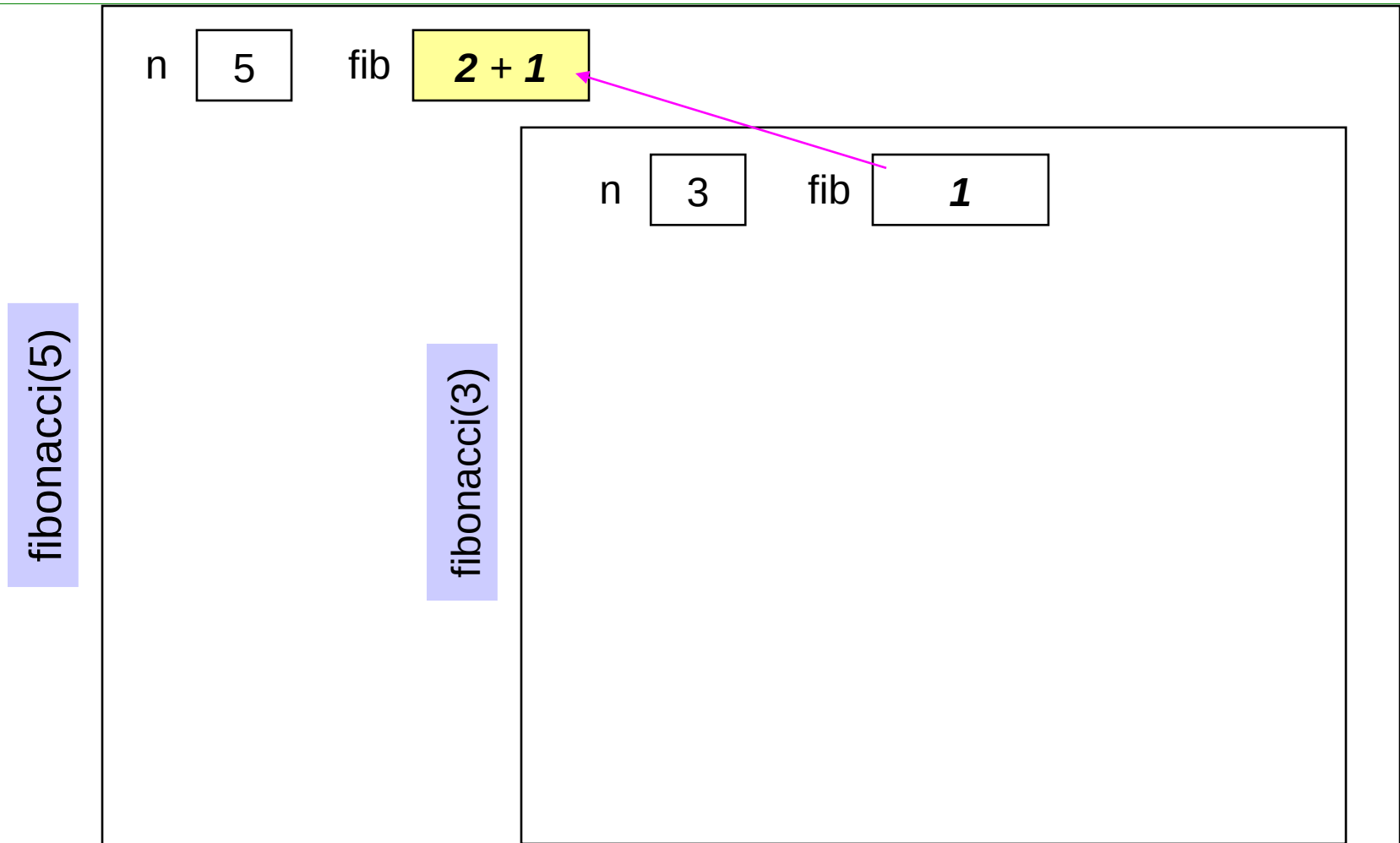
Traza de Ejecución (con cajas)



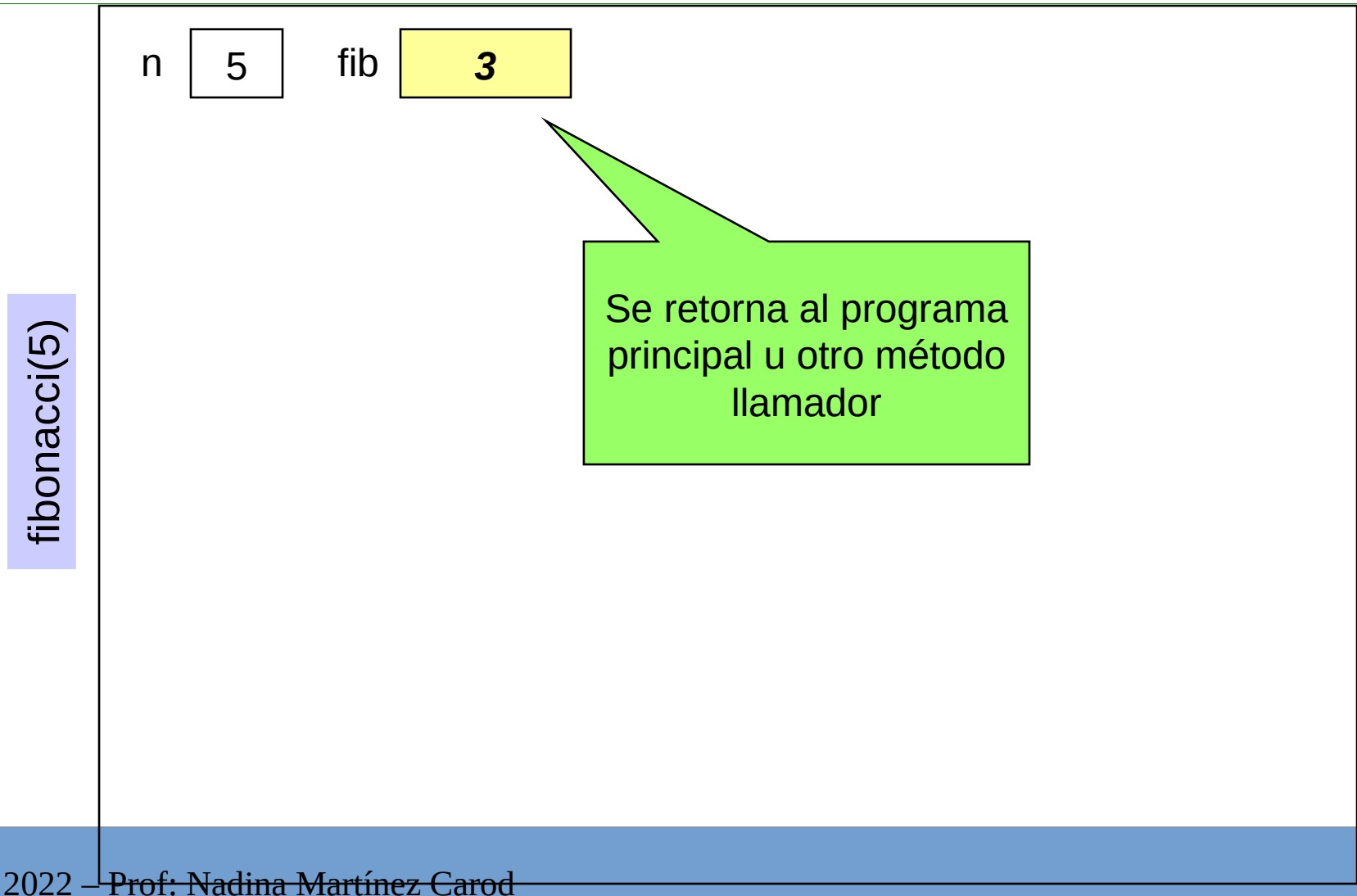
Traza de Ejecución (con cajas)



Traza de Ejecución (con cajas)



Traza de Ejecución (con cajas)



Traza de Ejecución (con Pila)

fibonacci(5)

```
public static int fibonacci(int n)
{
    int fib;
    if (n > 2) 1 2
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}
```

5	? + ...
n	fib

4	? + ...
5	? + ...
n	fib

3	? + ...
4	? + ...
5	? + ...
n	fib

2	1
3	? + ...
4	? + ...
5	? + ...
n	fib

Traza de Ejecución (con Pila)

fibonacci(5)

```
public static int fibonacci(int n)
{
    int fib;
    if (n > 2)
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}
```

2	1
3	1 + ...
4	? + ...
5	? + ...
n	fib

3	1 + ?
4	? + ...
5	? + ...
n	fib

1	0
3	1 + ?
4	? + ...
5	? + ...
n	fib

1	0
3	1 + 0
4	? + ...
5	? + ...
n	fib

3	1
4	? + ...
5	? + ...
n	fib

Traza de Ejecución (con Pila)

fibonacci(5)

```
public static int fibonacci(int n)
{  int fib;
   if (n > 2)
       fib = fibonacci(n-1) + fibonacci(n-2);
   else if (n == 2)
       fib = 1;
   else
       fib = 0;
   return fib;
}
```

3	1
4	1 + ...
5	? + ...

n fib

4	1 + ?
5	? + ...

n fib

2	1
4	1 + 1
5	? + ...

n fib

2	1
4	1 + 1
5	? + ...

n fib

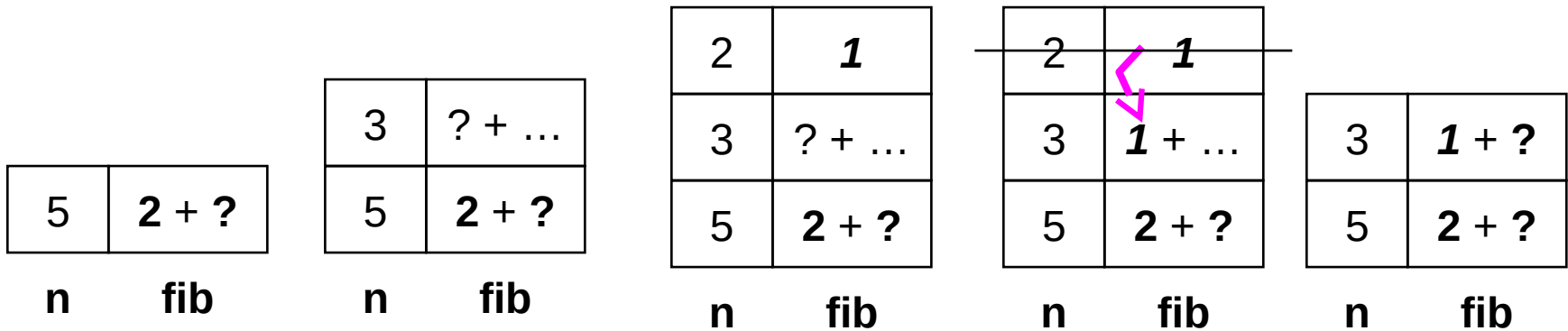
4	2
5	2 + ...

n fib

Traza de Ejecución (con Pila)

fibonacci(5)

```
public static int fibonacci(int n)
{   int fib;
    if (n > 2)
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}
```



Traza de Ejecución (con Pila)

fibonacci(5)

```
public static int fibonacci(int n)
{
    int fib;
    if (n > 2)
        fib = fibonacci(n-1) + fibonacci(n-2);
    else if (n == 2)
        fib = 1;
    else
        fib = 0;
    return fib;
}
```

Se retorna al programa principal u otro método llamador

